Algorithmique des graphes

Jean Goubault-Larrecq

25 janvier 2024

Résumé

Version 6 (courante) : 25 janvier 2024 [phrase incomplète supprimée, la table 4 était en double]. Version 5 : 14 décembre 2023 [corrigé argument de complexité de l'algorithme de Ford à files]. Version 4 : 12 décembre 2023 [la question de l'existence d'une source dans la section sur les tris topologiques n'était pas traitée]. Version 3 : 8 décembre 2023. Version 2 : 6 décembre 2023. Version 1 : 5 novembre 2023.

Table des matières

1 Introduction

Les graphes sont un outil fondamental de modélisation de nombreux systèmes informatiques, et au-delà. Ce texte présente quelques fondements des algorithmes sur les graphes, et sert de support à un cours de L3 donné à l'ENS Paris-Saclay.

Le but de ce texte, et du cours associé sont :

1. bien sûr, de présenter les principaux algorithmes du domaine, avec leurs analyses de complexité ;
2. mais surtout, de donner les *fondements mathématiques* sous-jacents à ces algorithmes.

Concernant ce deuxième point, rarement traité complètement, on citera notamment que c'est la théorie des parcours en profondeur de graphes qui permet de justifier la correction d'algorithmes comme ceux du tri topologique ou des composantes fortement connexes de Tarjan. Le code seul ne permet pas de comprendre ces algorithmes, et réciproquement, la théorie des parcours en profondeur est une théorie mathématique, qui s'étudie indépendamment des algorithmes — même si sa justification est, in fine, la correction des algorithmes.

Ce texte est une adaptation directe de plusieurs chapitres et sections du livre *Éléments d'algorithmique*, par Danièle Beauquier, Jean Berstel, et Philippe Chrétienne, que l'on dénotera familièrement par BBC. Il est disponible en `http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.pdf`. Les résultats de ce livre seront annoncés sous la forme **Proposition 4.5 (BBC, page 128)** ou **Lemme 4.6 (BBC, page 129)** par exemple. Le résultats propres à ce texte-ci suivront une numérotation simple, comme **Lemme 4.3** ou **Proposition 6.12**.

Pourquoi ce texte, alors ?

— La première raison est d'avoir sous la main ce qui est fait en cours, sans avoir à le chercher dans un livre plus long, et traitant de nombreux autres sujets.

— La deuxième raison est de rendre certains arguments du BBC plus rigoureux, et certains autres plus clairs. Le BBC est globalement très rigoureux, et sans doute davantage que la plupart des publications en matière d'algorithmique. Par exemple, une tendance naturelle chez tout algorithmicien est de faire des preuves d'algorithmes et de programmes « avec les mains », c'est-à-dire en invoquant des idées intuitives (le programme ne peut passer par ici puisqu'il est déjà passé par là), mais non fondées sur des définitions mathématiques. Le BBC est souvent à mi-chemin : il fournit une bonne abstraction mathématique des programmes présentés, mais l'abandonne parfois en cours de route pour des arguments « avec les mains ». Je céderai moi-même à cette tentation à partir de la section 9.6, où je serai un peu moins rigoureux quant à la preuve des programmes présentés (écrits en une notation pseudo-Caml, pseudo-Python, voire pseudo-C, comme partout ailleurs

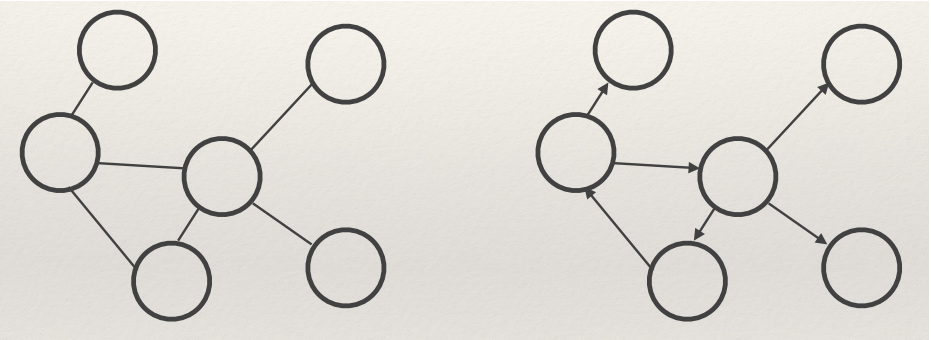


FIGURE 1 – Un graphe non orienté à gauche, un graphe orienté à droite

dans ces notes), tout en restant complètement formel en matière de preuve d'*algorithmes*, qu'on écrira alors sous forme plus abstraite de systèmes de transitions.

Par ailleurs, ce texte présente juste les éléments essentiels, et sa vocation est donc d'être raisonnablement concis. Il contient donc relativement peu d'intuitions.

2 Définitions de base

Avant de commencer, une note. Nous utiliserons la notation O dite de Landau. En informatique, on n'hésite pas à utiliser cette notation pour des fonctions de plusieurs variables, et il a été observé que cela est au mieux mal défini (Kalle Rutanen, *O-Notation in Algorithm Analysis*, rapport `arXiv:1309.3210` [cs.DS], 2016). Nous considérons que dire qu'une fonction g, disons de deux paramètres m et n, est en $O(f(m,n))$ signifie qu'il existe deux constantes $a, b > 0$ telles que pour tous $m, n \in \mathbb{N}$, $g(m,n) \leq af(m,n) + b$.

Il y a deux formes de graphes : les graphes orientés et les graphes non orientés, voir la figure 1.

Définition 2.1 *Un* graphe orienté*, ou simplement un* graphe*, est un couple* $G \stackrel{def}{=} (S, A)$*, où :* graphe orienté, graphe

— S *est un ensemble fini, dit de* sommets *;* sommets

— $A \subseteq S \times S$ *; c'est l'ensemble des* arcs *de* G*.* arcs

Étant donné un graphe (orienté) fixé $G \stackrel{\text{def}}{=} (S, A)$, on notera $u \to v$ pour dire qu'il y a un arc de u à v, autrement dit que $(u, v) \in A$; v est un *successeur* de u, et u est un *prédécesseur* de v dans G. successeur, prédécesseur

Définition 2.2 *Un* graphe non orienté *un un couple* $G \stackrel{def}{=} (S, A)$*, où :* graphe non orienté

— S *est un ensemble fini, dit de* sommets *;* sommets

— *A est une famille de paires $\{u, v\}$ (c'est-à-dire d'ensemble de cardinal exactement 2), appelées* arêtes. arêtes

La différence avec un graphe orienté est qu'une arête n'a pas de direction a priori. On notera u — v au lieu de $u \to v$ pour dire que $\{u, v\} \in A$, dans le contexte d'un graphe non orienté fixé. Les sommets u et v sont *adjacents*, ou *voisins*, dans G s'il existe une arête u — v dans G. adjacents voisins

On notera que dans un graphe non orienté tel que défini ci-dessus, il ne peut pas y avoir d'arête de cardinal 1, autrement dit de *boucle* u — u. Autoriser les boucles est une variante que l'on rencontre parfois de la notion de graphe non orienté. boucle

Dans ce texte, on s'intéressera principalement aux graphes *orientés* — à part en section 3 et en section 9.10 — est c'est pourquoi on les appellera le plus souvent, simplement, *graphes*. On peut toujours voir un graphe non orienté (S, A) comme le graphe orienté $(S, \{(u, v), (v, u) \mid \{u, v\} \in A\})$. orientés

Dans un graphe, il n'y a jamais plus d'un arc d'un sommet u à un sommet v. Il arrive de rencontrer dans la littérature une autre notion de graphe, que nous préférons appeler *multigraphe*, où A n'est plus un ensemble de couples, mais un ensemble abstrait d'arcs, avec deux fonctions $\partial_0, \partial_1 \colon A \to S$ associant à chaque arc son origine et son extrémité respectivement. La différence avec la définition précédente est qu'il peut y avoir plusieurs arcs entre deux sommets donnés. Nous n'utiliserons quasiment jamais cette notion, sauf à partir de la notion de graphe d'écart à la section 9.3. multigraphe

Il est standard de noter n le nombre de sommets d'un graphe et m son nombre d'arcs (ou d'arêtes, dans le cas d'un graphe non orienté). La complexité des algorithmes s'exprimera alors comme une fonction de n et de m, sans que l'on ait besoin de redire ce que n et m signifient.

On notera que l'on a toujours $m \leq n^2$. Donc une complexité en $O(mn)$ est meilleure en générale qu'un complexité en $O(n^3)$, par exemple. Ou bien, une complexité en $O(m+n^2)$ est en fait en $O(n^2)$. En revanche, on n'a pas nécessairement $n \leq m$: un sommet peut ne pas avoir de successeur.

Il existe au moins trois façons standard de représenter un graphe orienté en mémoire.

— La plus courante est la représentation par *listes de successeurs*. Les sommets du graphes sont numérotés de 1 à n et, en identifiant les sommets à leurs numéros, le graphe est représenté par un tableau à n entrées. Pour chaque sommet i, l'entrée numéro i du tableau est une liste des successeurs de i dans le graphe, pas nécessairement triée, mais *sans doublon*. Cette représentation utilise une place $\Theta(n + m)$. listes de successeurs

— La représentation par *matrice d'adjacence* représente le graphe comme une matrice $n \times n$; on trouve un 1 à la position (i, j) s'il y a un arc $i \to j$, sinon un 0. Cette représentation utilise une place $\Theta(n^2)$. matrice d'adjacence

— La représentation par *matrice d'incidence* représente le graphe comme une matrice $n \times m$; on n'en parlera pas ici. matrice d'incidence

En voyant chaque graphe non orienté comme un graphe orienté, la première représentation s'appelle alors représentation par *listes de voisins*. La représentation par matrice d'adjacence d'un graphe non orienté est une matrice symétrique avec des zéros sur la diagonale. listes de voisins

Définition 2.3 *Un* sous-graphe *d'un graphe orienté* $G \stackrel{def}{=} (S, A)$ *est un graphe de la forme* (S', A') *avec* $S' \subseteq S$ *et* $A' \subseteq A$. sous-graphe

Un sous-graphe induit *d'un graphe orienté* $G \stackrel{def}{=} (S, A)$ *est un graphe de la forme* $(S', A \cap S'^2)$ *avec* $S' \subseteq S$. *On dit qu'il est induit* par *le sous-ensemble de sommets* S'. sous-graphe induit

Un sous-graphe couvrant *d'un graphe orienté* $G \stackrel{def}{=} (S, A)$ *est un graphe de la forme* (S, A') *avec* $A' \subseteq A$. sous-graphe couvrant

3 Chemins, connexité, arbres non orientés

Définition 3.1 *Un* chemin *dans un graphe orienté* G *est une suite de sommets* $x_0 \to x_1 \to \cdots \to x_k$, *où* $k \in \mathbb{N}$. *Son* origine *est* x_0, *son* extrémité *est* x_k, *et sa* longueur *est* k. *On dit aussi qu'il s'agit d'un chemin* de x_0 à x_k. chemin origine extrémité longueur

On notera que la longueur k d'un chemin est son nombre d'arcs. Son nombre de sommets (pas nécessairement distincts) est $k + 1$.

On en déduit qu'un chemin dans un graphe non orienté est une suite de sommets $x_0 — x_1 — \cdots — x_k$. Vu la symétrie de la situation, on parlera de chemin *entre* x_0 *et* x_k plutôt que de x_0 à x_k ; ceci ne s'applique qu'aux graphes non orientés.

Dans la suite, je noterai parfois $\gamma \colon x \to^* y$ pour énoncer que γ est un chemin de x à y. Il m'arrivera aussi de ne pas nommer le chemin, et de juste parler d'un chemin $x \to^* y$, puis de former d'autres chemins, par exemple $x \to^* y \to z$. Il s'agit d'un abus de langage que je m'autoriserai lorsqu'il n'y aura pas de confusion possible, et qui signifie qu'étant donné un chemin γ de la forme $x = x_0 \to x_1 \to \cdots x_k = y$, on peut former (par exemple) le chemin $x = x_0 \to x_1 \to \cdots x_k = y \to z$, ce dernier étant abrégé en $x \to^* y \to z$.

3.1 Graphes connexes, composantes connexes

Définition 3.2 *Un graphe non orienté est* connexe *si et seulement si il existe un chemin entre n'importe quel couple de sommets.* connexe

Autrement dit, $G \stackrel{\text{def}}{=} (S, A)$ est connexe si et seulement si la clôture réflexive-transitive $—^*$ de — est la relation universelle S^2.

Dans tout graphe non orienté G, $—^*$ est une relation d'équivalence.

Définition 3.3 *Les* composantes connexes *d'un graphe non orienté G sont les classes d'équivalence de $—^*$.* composantes connexes

Autrement dit, les composantes connexes de G sont les *cliques maximales* de G, où une *clique* est un sous-ensemble non vide de sommets C tel que pour tous $u, v \in C$, on a $u — v$. clique

Lemme 3.1 (BBC, section 4.3.1, page 90) *Tout graphe non orienté connexe à $n \geq 1$ sommets a au moins $n-1$ arêtes ($m \geq n-1$).*

Démonstration. Par récurrence sur $n \geq 1$. Soit $G \stackrel{\text{def}}{=} (S, A)$ un graphe non orienté à n sommets. On choisit un sommet u, et on considère le sous-graphe induit H par $S \setminus \{u\}$. On énumère les composantes connexes $C_1, \ldots, C_k$ de H. Comme G est connexe, pour chaque $j \in \{1, \cdots, k\}$, il existe un chemin $u —^* v_j$ entre u et un sommet v_j de C_j. Ce chemin est de longueur non nulle car u n'est pas un sommet de C_j. Donc ce chemin est de la forme $u — u_j —^* v_j$. Comme $u \neq u_j$, u_j est un sommet de H, donc d'une composante connexe C_i. Comme C_i et C_j sont des classes d'équivalence pour $—^*$, et que $u_j —^* v_j$, on a donc $C_i = C_j$, et par suite, $i = j$. On a donc trouvé une arête (et pas juste un chemin) $u — u_j$ entre u et un sommet u_j de C_j, pour chaque $j \in \{1, \cdots k\}$.

Notons n_j le nombre de sommets de C_j. On a $n = 1 + \sum_{j=1}^{k} n_j$. Par hypothèse de récurrence, chaque C_j contient au moins $n_j - 1$ arêtes. Il y a k arêtes $u — u_j$, et en sommant, nous obtenons au moins $k + \sum_{j=1}^{k}(n_j - 1) = \sum_{j=1}^{k} n_j = n - 1$ arêtes dans G. □

3.2 Chaînes et cycles

Définition 3.4 *Une* chaîne *dans un graphe non orienté est un chemin $x_0 — x_1 — \cdots — x_k$ ($k \in \mathbb{N}$) dont les arêtes sont distinctes deux à deux. Un* cycle *est une chaîne non vide ($k \geq 1$) telle que $x_0 = x_k$.* chaîne cycle

On notera bien qu'une chaîne peut passer plusieurs fois par le même sommet ; seules les arêtes doivent être uniques. Un cycle $x_0 — x_1 — \cdots — x_k = x_0$ est *élémentaire* si et seulement si $x_1, \ldots, x_k$ sont distincts deux à deux (pas x_0, puisque $x_0 = x_k$). élémentaire

On observe le résultat trivial suivant.

Lemme 3.5 (D. Kőnig, graphes non orientés) *Si dans un graphe non orienté, il existe une chemin entre un sommet u et un sommet v, alors il existe un chemin de longueur minimale entre u et v, et tout chemin de longueur minimale entre u et v est une chaîne élémentaire.*

Démonstration. L'existence d'un chemin de longueur minimale est due à la bonne fondation de l'ordre usuel sur les entiers naturels. Si $u = x_0 — x_1 — \cdots — x_k = v$

est un chemin entre u et v, et si un sommet y apparaît deux fois, disons en tant que x_i et en tant que x_j, avec $i < j$, alors $x_0 — x_1 — \cdots — x_i = x_j — x_{j+1} — \cdots — x_k$ est un chemin strictement plus court. Un chemin de longueur minimale entre u et v ne peut donc mentionner chaque sommet qu'au plus une fois, et donc aussi chaque arête au plus une fois.

Un graphe non orienté est *sans cycle* si et seulement si il ne contient aucun cycle. sans cycle

Lemme 3.2 (BBC, section 4.3.1, page 91) *Tout graphe non orienté sans cycle à $n \geq 1$ sommets a au plus $n-1$ arêtes ($m \leq n-1$).*

Démonstration. On procède comme au lemme 3.1, par récurrence sur $n \geq 1$. Soit G un graphe non orienté à n sommets, sans cycle. Si $n = 1$, c'est évident, car G ne contient pas de boucle, donc $m = 0$. Supposons donc $n \geq 2$. On choisit un sommet u dans G, on note H le sous-graphe induit par $S \setminus \{u\}$, et on énumère les voisins $u_1, \ldots, u_k$ de u dans G. Ils sont tous dans H, puisque G n'a pas de boucle. De plus, les composantes connexes $C_1, \ldots, C_k$ de $u_1, \ldots, u_k$ (respectivement) dans H sont disjointes. En effet, si on avait deux indices $i \neq j$ dans $\{1, \cdots, k\}$ telles que les composantes connexes de u_i et u_j s'intersectent, elles seraient identiques, et il existerait donc un chemin $u_i —^* u_j$ dans H. Mais alors $u_i —^* u_j — u — u_i$ serait un cycle dans G.

L'union $C_1 \cup \cdots \cup C_k$ ne contient pas nécessairement tous les sommets de $H \setminus \{u\}$. Énumérons les composantes connexes $C_{k+1}, \ldots, C_{k+k'}$ restantes de H. Pour chaque $j \in \{1, \cdots, k+k'\}$, notons H_j le sous-graphe induit de H par C_j, et notons n_j son nombre de sommets. On a $n = 1 + \sum_{j=1}^{k+k'} n_j$. Le graphe H_j est lui aussi sans cycle, et $n_j \geq 1$, donc par hypothèse de récurrence il a au plus $n_j - 1$ arêtes. Mais les arêtes de G se découpent en :

- les k arêtes $u — u_j$, $j \in \{1, \cdots, k\}$;
- les au plus $\sum_{j=1}^{k+k'} (n_j - 1)$ arêtes des graphes H_j, $j \in \{1, \cdots, k+k'\}$;
- et aucune autre, puisque les H_j sont des composantes connexes de H.

Il y donc dans G au plus $k + \sum_{j=1}^{k+k'} (n_j - 1) \leq k + k' + \sum_{j=1}^{k+k'} (n_j - 1) = \sum_{j=1}^{k+k'} n_j = n - 1$ arêtes. □

3.3 Arbres non orientés

Définition 3.6 *Un* arbre non orienté *est un graphe connexe et sans cycle.* arbre non orienté

Cette notion est simplement appelée *arbre* dans le BBC, mais je préfère réserver ce terme pour les arbres orientés, une notion différente, quoique proche, que nous verrons en section 5.3. Voir la figure 2.

Proposition 3.3 (BBC, section 4.3.1, pages 91, 92) *Pour un graphe non orienté G à $n \geq 1$ sommets, les six propriétés suivantes sont équivalentes.*

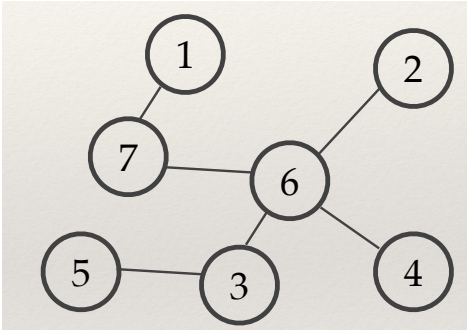


FIGURE 2 – Un arbre non orienté

1. *G est un arbre non orienté ;*
2. *G est connexe et a $n-1$ arêtes ;*
3. *G est sans cycle et a $n-1$ arêtes ;*
4. *il existe une unique chaîne entre deux sommets quelconques ;*
5. *G est minimal connexe, autrement dit il est connexe, mais enlever une arête quelconque de G détruit la connexité ;*
6. *G est maximal sans cycle, autrement dit il est sans cycle, mais ajouter une arête entre deux sommets distincts crée un cycle.*

Démonstration. Par le lemme 3.1 et le lemme 3.2, 1 implique 2 et 3.

On montre $2 \Rightarrow 3$ comme suit. Supposons que G soit connexe, avec $n \geq 1$ sommets, et ait $n-1$ arêtes, et un cycle $x_0 — x_1 — \cdots — x_k = x_0$, $k \geq 1$. On forme le graphe H à partir de G en supprimant une arête de ce cycle, par exemple $x_0 — x_1$. Le graphe H est toujours connexe : pour tout couple de sommets u et v, il existe une chaîne $u = y_0 — y_1 — \cdots — y_m = v$ entre u et v dans G par le lemme 3.5. Si aucune des arêtes de cette chaîne n'est l'arête supprimée $x_0 — x_1$, alors cette chaîne est une chemin dans H. Sinon, l'arête supprimée apparaît une et une seule fois dans $y_0 — y_1 — \cdots — y_m$, puisqu'il s'agit d'une chaîne, disons à la position i : autrement dit, $\{y_i, y_{i+1}\} = \{x_0, x_1\}$. Si $y_i = x_1$ et $y_{i+1} = x_0$, alors $y_0 — y_1 — \cdots y_i = x_1 — \cdots — x_k = x_0 = y_{i+1} — \cdots — y_m$ est un chemin entre u et v dans H. Si $y_i = x_0$ et $y_{i+1} = x_1$, alors $y_0 — y_1 — \cdots y_i = x_0 = x_k — x_{k-1} — \cdots — x_1 = y_{i+1} — \cdots — y_m$ est un chemin entre u et v dans H. Maintenant que nous avons montré que H est connexe, on observe qu'il a toujours n sommets, mais seulement $n-2$ arêtes, ce qui est impossible par le lemme 3.1.

On a aussi $3 \Rightarrow 2$. Supposons que G soit sans cycle et ait $n-1$ arêtes. Soient $C_1, \ldots, C_k$ ses composantes connexes, et notons n_i le nombre de sommets de C_i, $1 \leq i \leq k$. Comme G a au moins un sommet, $k \geq 1$. Chaque C_i est sans cycle, donc a au plus $n_i - 1$ arêtes par le lemme 3.2. Donc G a au plus $\sum_{i=1}^{k}(n_i - 1) = n - k$ arêtes. Mais il en a $n-1$, donc $n-1 \leq n-k$, donc $k \leq 1$. Or $k \geq 1$, donc $k = 1$. G est donc sa seule composante connexe, et est donc connexe.

On en déduit $3 \Rightarrow 1$: si 3 est vrai, alors 2 aussi, donc G est à la fois connexe et sans cycle.

Les points 1, 2, et 3 sont donc équivalents. Montrons qu'ils impliquent 4. Supposons donc que 1, 2, et 3 sont vérifiés. Comme G est connexe, tout couple de sommets u, v est relié par un chemin, donc par une chaîne par le lemme 3.5. Pour ce qui est de l'unicité, supposons qu'il existe deux sommets u et v de G et deux chaînes distinctes $\gamma_1 \colon u = x_0 — x_1 — \cdots — x_k = v$ et $\gamma_2 \colon u = y_0 — y_1 — \cdots — y_m = v$, et choisissons-les de sorte que la somme $k + m$ de leurs longueurs soit minimale. Autrement dit, on ne peut pas trouver u, v, et $\gamma_1 \neq \gamma_2$ de sorte à rendre $k + m$ strictement plus petit. La concaténation γ de γ_1 avec γ_2 (en ordre inverse) forme alors un cycle, ce qui est impossible par le point 3. Ceci démontre le point 4… au détail près qu'il faut vérifier que γ est bien un cycle ; c'est un point complètement éludé dans le BBC, et c'est là que la minimalité de $k+m$ va jouer.

- D'abord, on a $k + m \geq 1$: sinon, alors on aurait $k = m = 0$, donc $u = v$ et γ_1 et γ_2 seraient toutes les deux la chaîne de longueur nulle, ce qui est impossible car elles sont distinctes.
- Si $k = 0$, et donc $m \geq 1$, alors γ_2 est un cycle entre u et $v = u$, ce qui est impossible. De même, $m = 0$ est impossible.
- Les premières arêtes $x_0 — x_1$ et $y_0 — y_1$ (ce qui a un sens puisque $k, m \geq 1$) sont forcément distinctes, sinon $x_1 — \cdots — x_k$ et $y_1 — \cdots — y_m$ seraient deux chaînes distinctes de longueur totale $k + m - 2 < k + m$, entre les mêmes points $x_1 = y_1$ et $x_k = y_m = v$, contredisant la minimalité de $k+m$.
- Imaginons qu'il y ait une arête répétée dans γ. Comme γ_1 et γ_2 sont des chaînes, ceci n'est possible que si une arête $x_i — x_{i+1}$ de γ_1 coincide avec une arête $y_j — y_{j+1}$ de γ_2, et l'on vient de voir que nécessairement, i ou j est non nul. Par symétrie, c'est-à-dire en échangeant au besoin les rôles de γ_1 et de γ_2, supposons $i \geq 1$.
  - Si $x_i = y_j$ et $x_{i+1} = y_{j+1}$, alors $x_0 — x_1 — \cdots — x_i$ et $y_0 — y_1 — \cdots — y_j$ sont deux chaînes entre les mêmes couples de points $u = x_0 = y_0$ et $x_i = y_j$, et la somme de leurs longueurs est $i + j < k + m$; donc, par minimalité de $k+m$, ces deux chaînes sont égales. Mais ceci implique d'abord que $i = j$, et ensuite, comme $i \geq 1$, que $x_0 — x_1$ et $y_0 — y_1$ sont la même arête, ce qui est impossible.
  - Si $x_i = y_{j+1}$ et $x_{i+1} = y_j$, alors $x_0 — x_1 — \cdots — x_i$ et $y_0 — y_1 — \cdots — y_j — y_{j+1}$ sont deux chaînes entre les mêmes couples de points $u = x_0 = y_0$ et $x_i = y_{j+1}$, et la somme de leurs longueurs est $i+j+1 < k + m$; donc, par minimalité de $k + m$, ces deux chaînes sont égales. Mais ceci implique que $x_0 — x_1$ et $y_0 — y_1$ sont la même arête, ce qui est impossible.

Montrons $4 \Rightarrow 5$. Par le point 4, G est connexe. Si on pouvait enlever une arête $u — v$ de G sans casser la connexité, c'est qu'il y aurait un chemin, et donc

une chaîne (par le lemme 3.5) $u = x_0 — x_1 — \cdots — x_k = v$ ne passant pas par l'arête $u — v$. Donc $u = x_0 — x_1 — \cdots — x_k = v — u$ serait un cycle.

Montrons $5 \Rightarrow 6$. On suppose G minimal connexe. Si G avait un cycle, disons $x_0 — x_1 — \cdots — x_k = x_0$, alors on pourrait enlever l'arête $x_0 — x_1$, par exemple de G sans casser la connexité : dans tout chemin de G, on peut remplacer l'arête supprimée $x_0 — x_1$ par $x_0 = x_k — x_{k-1} — \cdots — x_1$. Si on ajoute une arête $u — v$ entre deux sommets distincts à G (donc cette arête n'est pas déjà dans G), alors on créerait un cycle. En effet, il existe déjà un chemin $u —^* v$ dans G, il ne contient pas $u — v$, qui n'est pas une arête de G, donc $u —^* v — u$ serait un cycle dans le graphe G plus l'arête $u — v$.

Finalement, montrons $6 \Rightarrow 1$. Si G est maximal sans cycle, il suffit de montrer que G est connexe. S'il ne l'était pas, il aurait au moins deux composantes connexes C_1 et C_2. Choisissons un sommet u dans C_1 et un sommet v dans C_2. Le graphe obtenu à partir de G en ajoutant l'arête $u — v$ n'a toujours pas de cycle. S'il y en avait un, disons $x_0 — x_1 — \cdots — x_k = x_0$ ($k \geq 1$), alors nécessairement l'une des arêtes de ce cycle est $u — v$, par exemple $x_i = u$, $x_{i+1} = v$. Aucune autre arête n'est égale à $u — v$, car un cycle est une chaîne. Les sommets x_0, x_1, …, x_i sont alors tous dans C_1, et les sommets x_{i+1}, …, x_k sont tous dans C_2. Mais ceci implique que $x_0 = x_k$ est à la fois dans C_1 et dans C_2, ce qui est impossible. □

4 Accessibilité

Définition 4.1 *Soit G un graphe orienté, et s et t deux sommets de G. On dit que t est* accessible *depuis s dans G si et seulement s'il existe un chemin $s = x_0 \to x_1 \to \cdots \to x_k = t$, $k \in \mathbb{N}$.* accessible

Ceci revient à dire que (s, t) est dans la clôture réflexive-transitive $\to^*$ de $\to$.

Définition 4.2 *Un chemin $x_0 \to x_1 \to \cdots \to x_k$ ($k \in \mathbb{N}$) est* élémentaire *si et seulement s'il ne contient aucun sommet en double : si $x_i = x_j$ alors $i = j$.* élémentaire

Contrairement à un cycle élémentaire, un chemin élémentaire peut être de longueur nulle ($k = 0$).

4.1 Chemins élémentaires

L'accessibilité est un problème fondamental en informatique. Pour décider de l'accessibilité de t depuis s, une solution naïve serait d'énumérer tous les chemins partant de s, jusqu'à en trouver un d'extrémité t, mais en présence de circuits (de chemins d'un sommet u au même sommet u, de longueur ≥ 1), il y a un nombre infini de chemins. Il suffit d'énumérer les chemins élémentaires, qui sont en nombre fini, grâce au résultat trivial suivant.

Lemme 4.3 (D. Kőnig, graphes orientés) *Si dans un graphe orienté, il existe un chemin d'un sommet s à un sommet t, alors il existe un chemin de longueur minimale de s à t, et tout chemin de longueur minimale de s à t est un chemin élémentaire.*

Ceci se démontre comme le lemme 3.5.

4.2 Caractérisation de l'ensemble des sommets accessibles

Énumérer les chemins élémentaires prend cependant un temps exponentiel en n, et il y a une bien meilleure solution.

Définition 4.4 (Reach**)** *Pour tout sommet s dans un graphe orienté $G \stackrel{def}{=} (S, A)$ donné, on note* $\text{Reach}_G(s)$*, ou bien simplement* $\text{Reach}(s)$ *si aucune ambiguïté quant au graphe G n'est à craindre, l'ensemble des sommets u accessibles depuis s dans G.*

On note $\texttt{succ}_G(u) \stackrel{def}{=} \{v \in S \mid u \to v\}$*, ou simplement* $\texttt{succ}(u)$*, l'ensemble des successeurs d'un sommet u de G dans G.*

Nous utiliserons aussi $\texttt{succ}(u)$ pour dénoter, de façon ambiguë, la *liste* des successeurs de u dans une représentation de G par listes de successeurs.

Lemme 4.5 *Pour tout sommet s d'un graphe G,* $\text{Reach}(s)$ *est le plus petit ensemble E de sommets tels que :*

1. $s \in E$,
2. *et pour tout $u \in E$,* $\texttt{succ}(u) \subseteq E$.

Démonstration. D'abord, $\text{Reach}(s)$ est un tel ensemble E. On a $s \in \text{Reach}(s)$ parce que le chemin de longueur nulle de s à s est un chemin, et si $u \in \text{Reach}(s)$, alors il y a un chemin $s \to^* u$, et alors pour tout $v \in \texttt{succ}(u)$, on a un chemin $s \to^* u \to v$.

Montrons que tout ensemble E vérifiant 1 et 2 contient $\text{Reach}(s)$. Pour ceci, il suffit de montrer que pour tout chemin $s = x_0 \to x_1 \to \cdots \to x_k$ partant de s, x_k est dans E. Ceci s'effectue par récurrence sur k. Si $k = 0$, $x_0 \in E$ par la condition 1. Si $k \geq 1$, par hypothèse de récurrence x_{k-1} est dans E, et comme $x_k \in \texttt{succ}(x_{k-1})$, la condition 2 nous permet d'affirmer que x_k est dans E.

4.3 Un algorithme d'accessibilité à base de liste de travail

Pour décider si t est accessible depuis s dans un graphe G, on calcule $\text{Reach}(s)$, puis on teste si t y appartient. En table 1, on trouvera un algorithme typique d'accessibilité, à base de *liste de travail*. La liste `work` contient à tout moment une liste de sommets dont on a découvert qu'ils étaient accessibles, mais qu'on

liste de travail

```
   fun reach_set (G as (S, succ), s) =
1      work := succ (s);
2      marked := {s};
3      while work≠[] do
4            let u::rest=work in (* on récupère le prochain sommet à traiter u *)
5                if u ∈ marked (* si u est déjà marqué, *)
6                   then work:=rest; (* alors dépiler u *)
7                else (work:=succ(u)@rest; (* sinon, dépiler u, empiler ses successeurs *)
8                      marked:=marked ∪ {u}); (* et marquer u *)
9      return marked;
   fun reach(G, s,t) = t ∈ reach_set(G, s);
```

TABLE 1 – Un algorithme de test d'accessibilité

n'a pas encore traités, au sens où l'on n'en a pas encore exploré les successeurs. L'opérateur `@` est la concaténation de listes. On suppose le graphe G donné depuis l'extérieur via sa fonction `succ` de liste de successeurs, et on s'autorise à utiliser un type de données d'ensemble fini (`marked`) ; ceci facilitera les raisonnements, et on verra comment s'en passer plus loin.

Pour en montrer la correction, on montre que les invariants suivants sont vérifiés à la ligne 4 (avant le `let`). On identifie les listes, comme `work`, aux ensembles d'éléments qu'elles contiennent, pour éviter de surcharger les notations. S est l'ensemble de sommets du graphe G, $\to$ est sa relation successeur, et `s` est le sommet passé en paramètre à `reach_set`, supposé dans S.

(Inv 1) $\texttt{marked} \cup \texttt{work} \subseteq S$;

(Inv 2) pour tout $u \in \texttt{marked} \cup \texttt{work}$, $\texttt{s} \to^* u$;

(Inv 3) $\texttt{s} \in \texttt{marked}$;

(Inv 4) pour tout $u \in \texttt{marked}$, $\texttt{succ}(u) \subseteq \texttt{marked} \cup \texttt{work}$.

Initialement, **(Inv 3)** est vrai car on a supposé que $\texttt{s} \in S$. Les lignes 1 et 2 assurent que, à la première entrée dans la boucle, $\texttt{work} = \texttt{succ}(\texttt{s})$ et $\texttt{marked} = \{\texttt{s}\}$, et les autres invariants sont donc initialement vrais.

On vérifie ensuite que si les invariants sont vérifiés à l'entrée du corps de la boucle, en ligne 4, et si $\texttt{work} \neq []$, alors ils sont rétablis en fin de corps de boucle (lignes 4–8). On note `work` sous la forme $\texttt{u} :: \texttt{rest}$, et on pose $\texttt{work}'$, $\texttt{marked}'$ les valeurs des variables `work`, `marked` à la fin du corps de la boucle. Si $\texttt{u} \in \texttt{marked}$, alors $\texttt{work}' = \texttt{rest}$, $\texttt{marked}' = \texttt{marked}$, et on a :

1. $\texttt{marked}' \cup \texttt{work}' \subseteq S$: car $\texttt{marked}' = \texttt{worked}$, $\texttt{work}' = \texttt{rest} \subseteq \texttt{work}$, et par **(Inv 1)** ;
2. pour tout $u \in \texttt{marked}' \cup \texttt{work}'$, $\texttt{s} \to^* u$: les éléments u de $\texttt{marked}' \cup \texttt{work}'$ sont tous dans $\texttt{rest} \subseteq \texttt{work}$, donc accessibles depuis s par **(Inv 2)** ;

3. $\mathtt{s} \in \mathtt{marked}'$: par **(Inv 3)**, puisque $\mathtt{marked}' = \mathtt{marked}$;
4. pour tout $u \in \mathtt{marked}'$, $\mathtt{succ}(u) \subseteq \mathtt{marked}' \cup \mathtt{work}'$: par **(Inv 4)**, tout élément u de $\mathtt{marked}'$ ($= \mathtt{marked}$) a tous ses successeurs dans $\mathtt{marked} \cup \mathtt{work}$. Cet ensemble vaut $(\mathtt{marked}' \cup \mathtt{work}') \cup \{\mathtt{u}\}$, donc les successeurs de u sont dans $\mathtt{marked}' \cup \mathtt{work}'$ — ce que l'on souhaite montrer — ou bien égaux à $\mathtt{u}$; mais on est dans le cas où $\mathtt{u} \in \mathtt{marked}$, et $\mathtt{marked}' = \mathtt{marked}$, donc u est bien dans $\mathtt{marked}' \cup \mathtt{work}'$.

Si $\mathtt{u} \notin \mathtt{marked}$, alors $\mathtt{work}' = \mathtt{succ}(\mathtt{u}) \cup \mathtt{rest}$, $\mathtt{marked}' = \mathtt{marked} \cup \{\mathtt{u}\}$, et on a :

1. $\mathtt{marked}' \cup \mathtt{work}' \subseteq S$: autrement dit, $\mathtt{succ}(\mathtt{u})$, $\mathtt{rest}$, $\mathtt{marked}$ sont inclus dans S et $\mathtt{u}$ est dans S. On a $\mathtt{u} \in S$ parce que $\mathtt{u} \in \mathtt{work} \subseteq S$ par **(Inv 1)** ; donc $\mathtt{succ}(\mathtt{u})$ est (bien défini et) inclus dans S, par définition de $\mathtt{succ}$. Pour ce qui est de $\mathtt{rest}$, $\mathtt{marked}$, on utilise $\mathtt{rest} \subseteq \mathtt{work}$ et **(Inv 1)**.
2. $\mathtt{s} \in \mathtt{marked}'$: car $\mathtt{marked} \subseteq \mathtt{marked}'$, et en utilisant **(Inv 2)**.
3. Pour tout $u \in \mathtt{marked}'$, $\mathtt{succ}(u) \subseteq \mathtt{marked}' \cup \mathtt{work}'$. On le vérifie comme suit. Si $u \in \mathtt{marked}$, alors par **(Inv 4)** on a $\mathtt{succ}(u) \subseteq \mathtt{marked} \cup \mathtt{work}$. Les successeurs de u sont donc dans $\mathtt{marked} \cup (\mathtt{rest} \cup \{\mathtt{u}\}) \subseteq \mathtt{marked}' \cup \mathtt{work}'$ (puisque $\mathtt{marked} \cup \{\mathtt{u}\} = \mathtt{marked}'$ et $\mathtt{rest} \subseteq \mathtt{work}'$). Si $u \in \mathtt{marked}' \setminus \mathtt{marked}$, alors $u = \mathtt{u}$, et on a $\mathtt{succ}(u) \subseteq \mathtt{work}'$ puisque $\mathtt{work}' = \mathtt{succ}(\mathtt{u}) \cup \mathtt{rest}$.

On a donc établi les invariants **(Inv 1)** à **(Inv 4)**. En sortie de boucle, à la ligne 9, ces invariants sont vrais, et en plus $\mathtt{work} = []$. On peut donc simplifier les invariants en :

1. $\mathtt{marked} \subseteq S$;
2. pour tout $u \in \mathtt{marked}$, $\mathtt{s} \to^* u$;
3. $\mathtt{s} \in \mathtt{marked}$;
4. pour tout $u \in \mathtt{marked}$, $\mathtt{succ}(u) \subseteq \mathtt{marked}$.

Les point 1 énonce que $\mathtt{marked}$ est un ensemble de sommets de G ; il contient $\mathtt{s}$ par le point 3 et est stable par $\succ$ par le point 4, donc il contient $\mathrm{Reach}(\mathtt{s})$ par le lemme 4.5. Finalement, il est inclus dans $\mathrm{Reach}(\mathtt{s})$ par le point 2. On en déduit le théorème de correction partielle suivant.

Proposition 4.6 *Si* $\mathtt{reach\_set}(G, s)$ *termine, alors l'ensemble qu'il retourne est exactement* $\mathrm{Reach}_G(s)$. *Si* $\mathtt{reach}(G, s, t)$ *termine, alors il retourne vrai si* $s \to^* t$ *dans* G, *et faux sinon.*

4.4 Terminaison et complexité

L'algorithme Reach termine, et même en temps linéaire en $n + m$, pour les raisons (informelles) suivantes. D'abord, un sommet une fois marqué le reste jusqu'à la fin de l'algorithme.

On pourrait penser que ceci implique qu'un sommet v, une fois dépilé de la liste de travail $\mathtt{work}$ (à la ligne 4), n'y sera jamais rempilé, mais c'est faux. En

revanche il n'y sera jamais rempilé deux fois via un même arc $u \to v$. Voici pourquoi. Disons qu'un sommet v est empilé *via l'arc* $u \to v$ si et seulement s'il est empilé à la ligne 7, avec $\mathtt{u} = u$ et $v \in \mathtt{succ(u)}$. Si l'on empile v via l'arc $u \to v$ une première fois, alors u sera immédiatement marqué à la ligne 8. Si l'on empile v via un arc $u' \to v$ ultérieurement, alors ce sera fait à la ligne 7, qui ne peut s'exécuter que si u' n'est pas marqué ; or, comme un sommet une fois marqué le reste jusqu'à la fin de l'algorithme, nécessairement $u' \neq u$.

On en déduit qu'on ne peut passer à la ligne 4 qu'au plus une fois par *arc* de G. Autrement dit, le nombre de fois où la boucle 4–8 est prise est majoré par m.

Le temps d'exécution du corps de la boucle n'est *pas* constant : la concaténation à la ligne 7 prend un temps proportionnel à $\#\mathtt{succ(u)}$, le nombre d'éléments dans la liste $\mathtt{succ(u)}$. (Si l'on représente les listes comme en Caml, une concaténation consiste à rajouter un à un tous les éléments de $\mathtt{succ}(u)$ en tête de la liste.) On pourrait donc majorer grossièrement le temps d'exécution de `reach_set` par m^2, mais on peut faire mieux.

Le coût $\#\mathtt{succ(u)}$ n'est à payer que pour les sommets `u` non marqués (test de la ligne 5). Pour ceux-ci, le temps pris par le corps de la boucle est proportionnel à $\#\mathtt{succ(u)}$ plus une constante, donc majoré par quelque chose de proportionnel à $\#\mathtt{succ}(u) + 1$. Pour les autres, le temps pris par le corps de la boucle est juste une constante. Or, comme tout sommet marqué le reste pour tout le reste de l'algorithme, le cas d'un sommet non marqué, de coût proportionnel à $\#\mathtt{succ(u)}+1$, n'est à payer qu'une fois par sommet `u` au plus. Le temps pris par `reach_set` est donc majoré par une quantité proportionnelle à à $\sum_{u \in S}(\#\mathtt{succ}(u) + 1) + m$, plus une constante pour les lignes 1 et 2, c'est-à-dire un $O(n + m)$: la somme $\sum_{u \in S} \#\mathtt{succ}(S)$ vaut m, et $\sum_{u \in S} 1 = n$.

On reverra souvent ce motif d'une boucle dont le coût d'un tour est proportionnel à $\#\mathtt{succ}(u) + 1$ pour des sommets non marqués u, et à 1 pour des sommets marqués, et l'on retiendra que la complexité d'une telle boucle est alors en $O(n + m)$.

On énonce le théorème. Nous en donnons une démonstration plus formelle, par invariants et variants, par la suite.

Théorème 4.7 *Pour tout sommet* `s` *d'un graphe orienté* G, `reach_set`$(G, \mathtt{s})$ *termine en temps* $O(n + m)$ *et calcule* $\mathrm{Reach}_G(\mathtt{s})$. *L'accessibilité est décidée par* `reach` *en temps* $O(n + m)$.

Démonstration. Pour formaliser l'argument de complexité ci-dessus, on va démontrer l'invariant supplémentaire :

(Inv 5) il existe une liste ℓ d'arcs de G, deux à deux distincts, telle que $\mathtt{work} = [v \mid (u, v) \in \ell]$ et pour tout arc $(u, v) \in \ell$, $u \in \mathtt{marked}$.

La liste ℓ est, intuitivement, la liste des arcs (u, v) *via lesquels* on a empilé v sur la liste `work`.

Comme d'habitude, cet invariant est à interpréter à la ligne 4. Il est initialement vrai, la liste ℓ étant la liste des couples $(\mathtt{s}, v)$, avec $v \in \mathtt{succ(s)}$, grâce à la ligne 1. De plus, $\mathtt{s} \in \mathtt{marked}$ grâce à la ligne 2. L'invariant est préservé par le corps de boucle. Ceci est laissé en exercice, et consiste à réécrire l'argument informel précédant l'énoncé du théorème.

A l'aide de `(Inv 5)`, on peut maintenant raisonner et considérer, à tout point du programme `reach_set`, l'ensemble E des arcs (u, v) qui sont dans ℓ, ou bien tels que $u \notin \mathtt{marked}$. Ce ou est exclusif, par l'invariant `(Inv 5)`. Appelons les arcs de ℓ *en attente*, et les arcs (u, v) avec $u \notin \mathtt{marked}$ *non examinés*. On va aussi considérer l'ensemble des sommets marqués `marked`. L'effet d'un tour de boucle est le suivant :

1. si $\mathtt{u} \in \mathtt{marked}$, la ligne 6 élimine un élément de `work`, et de façon correspondante élimine un arc en attente ; comme `marked` est inchangé, les arcs non examinés ne changent pas, et donc E a simplement un élément en moins ; d'autre part, `marked` ne change pas ;
2. si $\mathtt{u} \notin \mathtt{marked}$, alors la ligne 7 enlève un arc de ℓ et en rajoute $\#\mathtt{succ(u)}$ (à ℓ), mais comme $\mathtt{u} \notin \mathtt{marked}$, ces $\#\mathtt{succ(u)}$ arcs étaient des arcs non examinés, qui étaient donc déjà dans E ; donc E a simplement un élément en moins ; et `marked` a exactement un élément de plus, `u`.

Au premier passage à la ligne 4, E contient exactement tous les arcs de G. Comme son cardinal diminue de 1 à chaque tour de boucle, on effectue au maximum m tours de boucle. De plus, le temps pris par le corps de boucle dans le premier cas ci-dessus ($\mathtt{u} \in \mathtt{marked}$) est constant, et celui pris dans le deuxième cas est majoré par quelque chose de proportionnel à $\#\mathtt{succ(u)} + 1$, pour des sommets `u` différents à chaque fois (puisque $\mathtt{u} \notin \mathtt{marked}$, et que `marked` ne fait qu'augmenter, et strictement uniquement dans le cas 2), pour un temps total majoré par quelque de proportionnel à $\sum_{u \in S}(\#\mathtt{succ}(u)+1) = n+m$. Le temps utilisé par `reach_set` est donc un $O(m) + O(n + m) = O(n + m)$. □

4.5 Marquage par tableau de booléens

Dans la plupart des langages, on ne dispose pas d'un type de données « ensemble fini » permettant d'implémenter directement `marked`. On peut implémenter `marked` par un tableau de booléens `markp` : un sommet est dans `marked` si et seulement si $\mathtt{markp}[u] = \mathtt{true}$. Le code modifié de cette façon est donné en table 2.

Les différences, minimes dans l'ensemble, sont repérées par des commentaires. La seule différence important est la nouvelle ligne numéro 1,5, qui initialise le tableau `markp`, et qui impose un surcoût en complexité en $O(n)$. On pourrait améliorer la complexité de l'initialisation grâce à une procédure d'initialisation retardée (normalement vue en première partie du cours d'algorithmique), mais

```
   fun reach_set (G as (S, succ), s) =
1      work := succ (s);
1,5    for each u ∈ S do markp(u):=false; (* initialisation de markp *)
2      markp (s):=true; (* plutôt que : marked := {s}; *)
3      while work≠[] do
4            let u::rest=work in
5                if markp[u] (* plutôt que u ∈ marked *)
6                   then work:=rest;
7                else (work:=succ(u)@rest;
8                      markp[u]:=true; (* plutôt que marked:=marked ∪ {u} *)
9      return; (* plutôt que return marked *)
   fun reach(G, s,t) = begin reach_set (G, s); markp[t] end;
                       (* plutôt que t ∈ reach_set(G, s) *)
```

TABLE 2 – Un algorithme de test d'accessibilité, avec tableau de marques

c'est inutile : la complexité totale de l'algorithme d'accessibilité modifié reste en $O(n + m)$, malgré ce surcoût.

4.6 Autres structures de données pour la liste de travail

Au lieu d'une *liste* de travail `work`, qui fonctionne comme une pile (LIFO, ou last-in-first-out), on peut utiliser une *file* (FIFO, ou first-in-first-out), ou bien d'autres structures comme des files à priorité. En général, on peut remplacer toutes ces structures de données par une structure de donnée abstraite, du moment que l'on dispose de fonctions `empty` créant une structure de travail abstraite vide, `push` permettant d'ajouter un sommet à la structure de travail abstraite, `nonempty` testant si la structure de travail abstraite en argument est non vide, et `pop` permettant de récupérer et d'enlever un élément de la structure de travail abstraite : voir la table 3. On peut bien sûr implémenter `marked` par un tableau de bits, comme à la section 4.5.

On pourra se convaincre que l'algorithme de la table 3 est toujours correct, et termine toujours en temps $O(n+m)$, si les opérations `empty`, `push`, `nonempty` et `pop` sont elles-mêmes en temps constant.

L'unique différence est l'ordre dans lequel les sommets de G seront explorés. Avec une liste, comme à la table 1, le parcours est *en profondeur* (« depth-first »). Avec une file, le parcours est *en largeur* (« breadth-first »).

```
   fun reach_set (G as (S, succ), s) =
1      work := empty; for each u ∈ S do push (u, work);
2      marked := {s};
3      while nonempty(work)) do
4            let u=pop(work) in (* on dépile le prochain sommet à traiter u *)
5                if u ∉ marked (* si u n'est pas déjà marqué, *)
7                then (for each v ∈ succ(u) do
                                  push (v, work); (* empiler ses successeurs *)
8                     marked:=marked ∪ {u}); (* et marquer u *)
9      return marked;
   fun reach(G, s,t) = t ∈ reach_set(G, s);
```

TABLE 3 – Un algorithme de test d'accessibilité, avec une structure de travail abstraite `work`

5 Parcours en profondeur

5.1 Circuits

Définition 5.1 *Un* circuit *dans un graphe G est une chemin $x_0 \to x_1 \to \cdots \to x_k$ non vide ($k \geq 1$) et tel que $x_0 = x_k$.*

circuit

Un graphe orienté acyclique*, ou* DAG *(pour « directed acyclic graph ») est un graphe sans circuit.*

graphe orienté acyclique

DAG

Imaginons que l'on nous donne un graphe (orienté) G, et que l'on nous demande si G est un DAG. On sait tester l'accessibilité en temps $O(n+m)$, et on pourrait donc simplement énumérer les arcs $u \to v$ de G, et tester si u est accessible depuis s. Ceci prendrait un temps $O((n+m)^2)$. On peut faire bien mieux à l'aide de parcours de graphes. Les parcours *en profondeur* ont une structure mathématique bien particulière qui les rendront intéressants pour de nombreuses autres applications.

Avant d'explorer les profondeurs, on va utiliser une astuce permettant de supposer que les graphes qui nous intéresseront auront une source, et définir une notion d'arbres orientés, qui sera fondamentale dans la suite.

5.2 Sources

Définition 5.2 *Une* source *d'un graphe orienté G est un sommet à partir duquel tout sommet est accessible.*

source

Un graphe orienté n'a pas nécessairement de source. Étant donné un graphe orienté $G \stackrel{\text{def}}{=} (S, A)$, on peut former un nouveau graphe G_* en lui adjoignant un nouveau sommet $*$, et de nouveaux arcs $* \to u$, $u \in S$. Le sommet $*$ est alors une

source de G_*. Dans la suite de cette section, et sauf exception, on supposera que $G \stackrel{\text{def}}{=} (S, A)$ est un graphe orienté d'unique source $*$. (Ceci simplifie le cas général traité dans le BBC.)

En pratique, cette adjonction d'un nouveau sommet $*$ est effectuée virtuellement : les algorithmes de graphes sont réécrits de sorte à simuler leur exécution sur G_*, sans le construire effectivement.

5.3 Arbres orientés

Définition 5.3 *Un* arbre *(orienté) est un triplet $(S, *, p)$ où $* \in S$, p est une application de $S \setminus \{*\}$ dans S, et pour tout $u \in S$, il existe un $n \in \mathbb{N}$ tel que $p^n(u) = * : *$ est la* racine *de l'arbre, $p(u)$ est le* prédécesseur *de u, et seule la racine n'a pas de prédécesseur.*

arbre
racine
prédécesseur

Ceci n'est pas (du tout) la définition d'un arbre du BBC, même s'il y a des liens.

Un arbre peut être vu comme un graphe orienté (S, A) avec source $*$, où $A \stackrel{\text{def}}{=} \{(p(u), u) \mid u \in S \setminus \{*\}\}$. Il y a alors un *unique* chemin $* = p^n(u) \to p^{n-1}(u) \to \cdots \to p(u) \to u$ de la racine $*$ vers n'importe quel sommet u.

Lemme 5.4 *Dans un arbre $T \stackrel{def}{=} (S, *, p)$:*

1. *La relation $\to_T^*$ est une relation d'ordre.*
2. *Si $u \to_T^* v$ alors il existe un unique chemin de u à v dans l'arbre.*
3. *Deux sommets quelconques u et v de S ont une borne inférieure $u \wedge v$ pour $\to_T^*$, appelée leur* ancêtre commun le plus proche.
4. *Si $u \to_T^* w$ et $v \to_T^* w$, alors u et v sont* comparables*, c'est-à-dire $u \to_T^* v$ ou $v \to_T^* u$.*
5. *Pour tous sommets u, v, s'il existe un chemin de u à v et $u \neq v$, alors il existe un chemin de u à $p(v)$ (et un arc de $p(v)$ à v).*

ancêtre commun le plus proche
comparables

Démonstration. On démontre d'abord que : $(*)$ si $p^m(u)$ est défini et est égal à u, alors $m = 0$. Sinon, $p^{km}(u)$ serait lui aussi défini et égal à u pour tout $k \in \mathbb{N}$. Ceci impliquerait que $p^\ell(u)$ serait défini pour une infinité de valeurs de ℓ. Or il existe un entier n tel que $p^n(u) = *$, et donc tel que $p^\ell(u)$ ne soit défini pour aucun $\ell \leq n + 1$, ce qui est impossible. Donc $m = 0$.

1. La réflexivité et la transitivité sont évidentes. Supposons $u \to_T^* v$ et $v \to_T^* u$. Donc $u = p^m(v)$ et $v = p^{m'}(u)$ pour deux entiers m et m'. On en déduit $p^{m+m'}(u) = u$, et donc, par $(*)$, que $m + m' = 0$, en particulier $u = v$.

2. Si $u \to_T^* v$, alors $u = p^m(v)$ pour un certain entier m. S'il existait un autre chemin de u à v, c'est que u serait aussi égal à $p^{m'}(v)$ pour un $m' \neq m$. Par symétrie, disons $m' > m$. On aurait alors $p^{m'-m}(u) = u$. Donc $m' = m$, par $(*)$: contradiction.

3. Par le point 2, on a deux chemins uniques $* = p^m(u) \to_T p^{m-1}(u) \to_T \cdots \to_T p(u) \to_T u$ et $* = p^n(v) \to_T p^{n-1}(v) \to_T \cdots \to_T p(v) \to_T v$. Leur plus grand préfixe commun est un chemin de $*$ à un sommet w tel que $w \to_T^* u$ et $w \to_T^* v$. Plus précisément, on a $w \stackrel{\text{def}}{=} p^{m-k}(u) = p^{n-k}(v)$, où k est l'entier le plus grand ($\leq \min(m, n)$) tel que $p^{m-k}(u) = p^{n-k}(v)$.

Si w' est n'importe quel sommet tel que $w' \to_T^* u$ et $w' \to_T v$. Alors w' s'écrit à la fois $p^i(u)$ et $p^j(v)$. De plus, il existe un entier k tel que $p^k(w) = *$. Donc le chemin $* = p^k(w) \to_T p^{k-1}(w) \to_T \cdots \to_T p(w) \to_T w$ est un préfixe de l'unique chemin de $*$ à w. En particulier, $w' \to_T^* w$.

4. Soient m et n des entiers tels que $u = p^m(w)$ et $v = p^n(v)$. Si $m \geq n$, alors $u = p^{m-n}(v)$ donc $u \to_T^* v$. Sinon, $v \to_T^* u$.

5. Il y a en fait un unique chemin $u = p^n(v) \to_T p^{n-1}(v) \to_T \cdots \to_T p(v) \to_T v$ de u à v, par le point 2. Comme $u \neq v$, on a $n \geq 1$, ce qui nous permet de conclure. □

Les définitions suivantes seront utiles dans la suite.

Définition 5.5 *Un* descendant *d'un sommet u dans un arbre T est un sommet v tel que $u \to_T^* v$; on dira alors aussi que u est un* ascendant *de v dans T.* descendant ascendant

On notera $D_T(u)$ l'ensemble des descendants de u dans l'arbre T.

Définition 5.6 *Un* arbre couvrant *de G est un arbre qui forme un sous-graphe de G, et qui contient tous les sommets de G.* arbre couvrant

C'est juste un arbre, qui est en même temps un sous-graphe couvrant de G. La racine d'un arbre couvrant est nécessairement une source de G (donc son unique source $*$, vu notre hypothèse de départ sur G).

5.4 Parcours

On rappelle que l'on suppose que G a une source fixée $*$. Un parcours, comme défini ci-dessous, est une façon de définir formellement la liste des sommets parcourus dans un algorithme, comme `reach_set` par exemple. Il est important de pouvoir le définir sans dépendance à un algorithme.

Définition 5.7 *Un* parcours *de G est une liste de sommets $L \stackrel{def}{=} [u_1, \cdots, u_n]$ de G telle que :* parcours

- *chaque sommet de G apparaît exactement une fois dans L ;*
- *$u_1 = *$;*
- *chaque sommet de L, sauf le premier, est successeur (dans G) d'un élément qui le précède dans L : autrement dit, pour tout j ($1 < j \leq n$), il existe un i ($1 \leq i < j$) tel que $u_i \to u_j$.*

L'unique élément qui n'a pas de prédécesseur doit donc être le premier ; autrement dit, on doit avoir $u_1 = *$.

Étant donné un parcours $L \stackrel{\text{def}}{=} [u_1, \cdots, u_n]$ de G, on notera L_k $(0 \leq k \leq n)$ le préfixe $[u_1, \ldots, u_k]$ formé des k premiers éléments de L. On a $L_n = L$. Les L_k, $1 \leq k \leq n$, sont des parcours partiels, au sens suivant.

Définition 5.8 *Un* parcours partiel *de G est une liste L_* de sommets $[u_1, \cdots, u_k]$ où $1 \leq k \leq n$, où $u_1 = *$, où chaque sommet de G apparaît* au plus *une fois dans L_*, et où chaque sommet sauf le premier est successeur d'un élément qui le précède dans L_*.* parcours partiel

Remarque 5.9 *Un parcours partiel à n éléments (où n est le nombre de sommets du graphe) est un parcours.*

Si pour chaque indice j $(1 < j \leq n)$, on *choisit* un i $(1 \leq i < j)$ tel que $u_i \to u_j$, on dira que $u_i \to u_j$ est un *arc de liaison*. On peut organiser les choix des arcs de liaison (un pour chaque j) comme une fonction $p \colon S \setminus \{*\} \to S$, qui à u_j associe *un* u_i avec $i < j$ tel que $u_i \to u_j$. Un tel choix n'est pas unique. Mais, étant donné un parcours L de G, tout choix d'une telle fonction p définit un arbre couvrant de G. On obtient donc : arc de liaison

Proposition 4.5 (BBC, section 4.4.4, page 107, modifiée) *Tout choix d'arcs de liaison d'un parcours de G constitue un arbre couvrant de G.*

Remarque 5.10 *C'est un arbre, et pas une forêt comme dans le BCC, parce que G a une source $*$. Si l'on ne suppose pas que G a une source, on définit un parcours de G comme étant une liste L telle que $* :: L$ soit un parcours de G_*. Tout choix d'arcs de liaison d'un tel parcours constitue un arbre couvrant de G_*, donc une* forêt couvrante *de G après en avoir supprimé la racine $*$. (Une forêt est une union disjointe finie d'arbres.)* forêt couvrante

Définition 5.11 *Un* sommet fermé *d'un parcours partiel L_* est un sommet de L_* dont tous les successeurs (dans G) sont aussi dans L_*.* sommet fermé

Un sommet ouvert *de L_* est un sommet de L_* qui a au moins un successeur qui n'est pas dans L_*.* sommet ouvert

Remarque 5.12 *Dans un parcours (non partiel) L, tout sommet est fermé. Étant donné un parcours partiel L_*, tout sommet de G est soit ouvert dans L_*, soit fermé dans L_*, soit hors de L_*.*

Lemme 5.13 *Soit $L_* \stackrel{def}{=} [u_1, \cdots, u_k]$ un parcours partiel de G, $1 \leq k \leq n$. Si $k < n$, alors L_* contient un sommet ouvert.*

Démonstration. Supposons le contraire. L'ensemble $\{u_1, \cdots, u_k\}$ des sommets de L_* forme alors une famille de sommets de G contenant $*$ et stable par successeurs, puisque tous ses sommets sont fermés dans L_k. Cet ensemble contient donc l'ensemble Reach$(*)$ des sommets accessibles depuis $*$, par le lemme 4.5. Or Reach$(*)$ vaut S tout entier, par définition de la source $*$. Donc il y a n éléments dans Reach$(*) = \{u_1, \cdots, u_k\}$, ce qui est impossible puisque $k < n$. □

Corollaire 5.14 *Il existe au moins un parcours L de G.*

Démonstration. Par récurrence sur k, on montre que pour tout k avec $1 \leq k \leq n$, il existe un parcours partiel de G. Pour $k = 1$, $[*]$ est un tel parcours partiel. Si $1 < k \leq n$, par hypothèse de récurrence il existe un parcours partiel $L_* \stackrel{\text{def}}{=} [u_1, \cdots, u_{k-1}]$ à $k-1$ éléments de G. Par le lemme 5.13, L_* contient un sommet ouvert u_i $(1 \leq i < k)$. Par définition, il existe un arc $u_i \to u_k$, où u_k est un sommet de G qui n'est pas dans L_*. Donc $[u_1, \cdots, u_{k-1}, u_k]$ est un parcours partiel de longueur k. □

Si L est un parcours de G, il existe donc pour chaque j $(1 < j \leq n)$ un *plus petit* indice i $(1 \leq i < j)$ tel que u_i soit un sommet ouvert de L_{j-1}. On dit que L est un *parcours en largeur* s'il y a un arc de chacun de cet u_i vers u_j, pour chaque j. On choisit alors (u_i, u_j), avec i choisi ainsi, comme arc de liaison d'extrémité u_j, pour chaque j. L définit un arbre couvrant de G, par $p(u_j) \stackrel{\text{def}}{=} u_i$ (pour chaque j, i étant le plus petit tel que u_i soit ouvert dans L_{j-1}). On dit usuellement, mais plus informellement (voir le BBC) : dans un parcours en largeur, l'origine de tout arc de liaison est le *premier* sommet ouvert déjà visité.

parcours en largeur

Remarque 5.15 *Dans le cas d'un graphe de la forme G_*, où il y a un arc $* \to u$ pour tout sommet u autre que $*$, la notion de parcours en largeur est triviale : tout parcours est en largeur. Elle l'est moins pour d'autres graphes.*

Lemme 5.16 *Soit $L \stackrel{def}{=} [u_1, \cdots, u_n]$ un parcours de G. On suppose un choix d'arcs de liaison effectué, ce qui définit un arbre couvrant T de G.*

1. *Tout sommet u de G s'écrit u_i pour un unique i $(1 \leq i \leq n)$.*
   On appelle i le rang *$r(u)$ de u dans L.*
2. *Tout descendant v d'un sommet u dans T satisfait $r(v) \geq r(u)$.*
3. *Soit $1 \leq i \leq j \leq k \leq n$. Si u_i est un sommet de L_j qui est ouvert dans L_k, alors il est ouvert dans L_j.*

rang

Démonstration. 1. Évident.

2. Il suffit de le démontrer lorsque $u = p(v)$, le résultat s'en déduisant par récurrence sur la longueur de l'unique chemin de u à v dans T. Soit $j \stackrel{\text{def}}{=} r(v)$, c'est-à-dire $u_j = v$. Le sommet $u = p(v)$ est défini comme un u_i avec $i < j$ satisfaisant certaines conditions. Mais alors $r(u) = i < j = r(v)$.

3. Si u_i était fermé dans L_j, tous ses successeurs (dans G) seraient des u_ℓ avec $\ell \leq j$. En particulier, $\ell \leq k$, et donc u_i serait fermé dans L_k. □

5.5 Parcours en profondeur

Dans un parcours en profondeur, c'est le *dernier* sommet ouvert déjà visité qui est l'origine de tout arc de liaison. Autrement dit,

Définition 5.17 *Soit $L \stackrel{def}{=} [u_1, \cdots, u_n]$ un parcours de G. L est un* parcours en profondeur *si et seulement, pour chaque j ($1 < j \leq n$), le* plus grand *indice i ($1 \leq i < j$) tel que u_i soit un sommet ouvert de L_{j-1} est tel que $u_i \to u_j$.*

parcours en profondeur

On rappelle qu'un tel i existe toujours par le lemme 5.13. On définit ces arcs $u_i \to u_j$ comme étant les arcs de liaison, et ceci définit de nouveau un arbre couvrant de G.

Nous fixons désormais un parcours en profondeur $L \stackrel{\text{def}}{=} [u_1, \cdots, u_n]$ de G (avec $u_1 = *$), et nous notons $T \stackrel{\text{def}}{=} (S, *, p)$ l'arbre couvrant associé. Un *descendant* d'un sommet u est un sommet v accessible depuis u dans T, c'est-à-dire tel que $u \to_T^* v$, autrement dit $p^n(v) = u$ pour un certain $n \in \mathbb{N}$. On dit aussi que u est un *ascendant* de v dans T.

descendant

ascendant

La proposition suivante n'est vraie que des parcours en profondeur, et est le point clé.

Proposition 5.18 *Pour tout sommet u de G, l'ensemble des descendants $D_T(u)$ de u dans T est un* intervalle *de L, c'est-à-dire un ensemble de la forme $\{u_i, u_{i+1}, \cdots, u_j\}$, où $i = r(u)$. On appelle j le* temps de fin *$fin(u)$ de u dans L.*

intervalle

temps de fin

De plus, $j = fin(u)$ est le plus petit entier supérieur ou égal à i tel que tous les sommets u_i, u_{i+1}, …, u_j soient fermés dans L_j.

Démonstration. On note d'abord que, par le lemme 5.16 (2), tout descendant de u a un rang plus grand ou égal à $r(u)$. L'ensemble des descendants $D_T(u)$ de u dans T est donc un sous-ensemble de $\{u_i, u_{i+1}, \cdots, u_n\}$, où $i \stackrel{\text{def}}{=} r(u)$. Il contient aussi $u_i = u$.

Soit j le plus petit entier supérieur ou égal à i tel que tous les sommets u_i, u_{i+1}, …, u_j soient fermés dans L_j. Un tel j existe (et est inférieur ou égal à n) car tous les sommets sont fermés dans L_n. On montre que tous les sommets u_k ($i \leq k \leq j$) sont dans $D_T(u)$, par récurrence sur $k - i$:

- Si $k - i = 0$, $u_i = u$ est dans $D_T(u)$.
- Sinon, par minimalité de j, et comme $k - 1 < j$, il existe un sommet parmi u_i, u_{i+1}, …, u_{k-1} qui soit ouvert dans L_{k-1}. Par définition d'un parcours en profondeur, l'arc de liaison $u_\ell \to_T u_k$ est tel que ℓ est maximal parmi les indices tels que u_ℓ soit ouvert dans L_{k-1}. Ceci implique $\ell \geq i$. Ainsi, on peut utiliser l'hypothèse de récurrence, d'où l'on déduit que $u_\ell \in D_T(u)$. On a donc un chemin de u à u_ℓ dans T, auquel on peut ajouter l'arc de liaison $u_\ell \to_T u_k$, et on en conclut que u_k est lui aussi dans $D_T(u)$.

Pour montrer que $D_T(u)$ vaut exactement $\{u_i, u_{i+1}, \cdots, u_j\}$, on raisonne par l'absurde. Si ce n'est pas le cas, il existe un sommet u_k, avec $j < k \leq n$, dans $D_T(u)$. Choisissons k minimal avec cette propriété. Comme $k > j \geq i$, u_k est différent de $u_i = u$. Donc le prédécesseur $u_\ell \stackrel{\text{def}}{=} p(u_k)$ de u_k dans T est bien défini. De plus, l'unique chemin de u à u_k dans T se décompose en un chemin de u à u_ℓ (dans T), suivi de l'arc de liaison $u_\ell \to_T u_k$ (lemme 5.4 (5)). Il s'ensuit que u_ℓ est aussi dans $D_T(u)$.

— Il est impossible que $\ell > j$, à cause de la minimalité de k.

— Si $i \leq \ell \leq j$, le sommet u_ℓ, qui est ouvert dans L_{k-1} par la définition du parcours en profondeur, serait aussi ouvert dans L_j, par le lemme 5.16 (3). C'est impossible aussi, parce que tous les sommets u_i, u_{i+1}, …, u_j sont fermés dans L_j.

— Finalement, le cas $\ell < i$ est impossible lui aussi, puisque l'existence d'un chemin dans T de $u = u_i$ à u_ℓ implique $i \leq \ell$, par le lemme 5.16 (2). □

Proposition 5.19 *Pour tout couple de sommets u et v dans G, les trois propriétés suivantes sont équivalentes :*

1. *v est un descendant de u dans T ;*
2. *$[r(v), fin(v)] \subseteq [r(u), fin(u)]$;*
3. *$r(u) \leq r(v)$ et $fin(v) \leq fin(u)$.*

Démonstration. 1 ⇒ 2. Comme v est un descendant de u dans T, $D_T(v) \subseteq D_T(u)$. Par la proposition 5.18, $D_T(v) = \{u_{r(v)}, u_{r(v)+1}, \cdots, u_{fin(v)}\}$ et $D_T(u) = \{u_{r(u)}, u_{r(u)+1}, \cdots, u_{fin(u)}\}$. Donc les indices $r(v)$, $r(v)+1$, …, $fin(v)$ sont tous dans l'ensemble $\{r(u), r(u)+1, \cdots, fin(u)\}$.

Il est clair que 2 et 3 sont équivalents.

3 ⇒ 1. Par la proposition 5.18, $D_T(u) = \{u_{r(u)}, u_{r(u)+1}, \cdots, u_{fin(u)}\}$. Comme $r(u) \leq r(v) (\leq fin(v)) \leq fin(u)$, $v = u_{r(v)}$ est dans $D_T(u)$. □

Proposition 5.20 *Pour tout couple de sommets u et v dans G, tels qu'aucun des deux n'est descendant de l'autre dans T, les intervalles $[r(u), fin(u)]$ et $[r(v), fin(v)]$ sont disjoints.*

Démonstration. S'ils ne l'étaient pas, il y aurait un indice i dans les deux. Par la proposition 5.18, u_i serait donc dans $D_T(u)$ et dans $D_T(v)$, autrement dit $u \to_T^* u_i$ et $v \to_T^* u_i$. Par le lemme 5.4 (4), u et v seraient comparables dans $\to_T^*$, ce qui contredirait l'hypothèse. □

Étant données deux sommets u_i et u_j, avec $i < j$, on n'a donc que deux possibilités : soit u_j est un descendant de u_i dans T (et $[i, fin(u_i)]$ contient $[j, fin(u_j)]$), soit les intervalles $[i, fin(u_i)]$ et $[j, fin(u_j)]$ sont disjoints, et donc en particulier $fin(u_i) < j$.

Ceci mène à la classification suivante des arcs $u \to v$ de G :

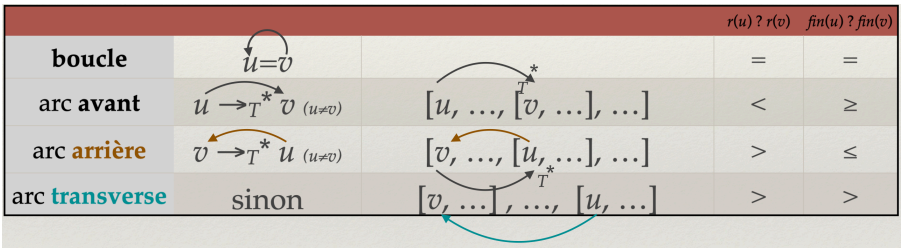


TABLE 4 – La classification des arcs par rapport à un parcours en profondeur

— c'est une *boucle* si $u = v$; boucle

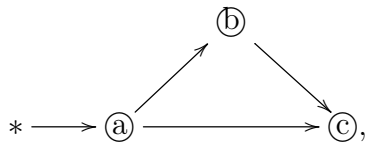
— c'est un *arc avant* si $u \to_T^* v$ et $u \neq v$; arc avant

— c'est un *arc arrière* si $v \to_T^* u$ et $u \neq v$; arc arrière

— c'est un *arc transverse* sinon. arc transverse

La table 4 résumé les différents cas de façon visuelle.

Remarque 5.21 *Parmi les arcs avants, on trouve bien sûr les arcs $u \to_T v$ de l'arbre T. Mais il y en d'autres. Par exemple, dans le graphe :*



avec comme parcours en profondeur $[*, ⓐ, ⓑ, ⓒ]$, l'arc ⓐ → ⓒ est un arc avant qui n'est pas un arc de l'arbre T.

Lemme 4.6 (BBC, section 4.4.4, page 108) *Pour tout arc transverse $u \to v$, $r(u) > r(v)$.*

Démonstration. Imaginons que $r(u) \leq r(v)$, et donc $r(u) < r(v)$ puisque $u \neq v$, un arc transverse n'étant pas une boucle. Par la proposition 5.18, Notons $i \stackrel{\text{def}}{=} r(u)$, $k \stackrel{\text{def}}{=} r(v)$, de sorte que $u = u_i$, $v = u_k$, et $i < k$.

Par la première partie de la proposition 5.18, $D_T(u)$ est un intervalle $\{u_i, u_{i+1}, \cdots, u_j\}$, avec $j = fin(u)$. De même, $D_T(v)$ est un intervalle $\{u_k, u_{k+1}, \cdots, u_\ell\}$, avec $\ell = fin(v)$. Les intervalles $[i, j]$ et $[k, \ell]$ sont disjoints par la proposition 5.20. Comme $i < k$, on en déduit $j < k$.

Mais alors, l'arc $u \to v$ (c'est-à-dire $u_i \to u_k$) montre que u_i est un sommet ouvert dans L_j, ce qui contredit la deuxième partie de la proposition 5.18. □

Note 5.22 *La démonstration dans le BBC ne me satisfait pas, pour deux raisons. En premier, elle utilise un raisonnement temporel (« lorsque le sommet x*

est visité, le sommet y ne l'est pas encore ») qui est, au mieux, une bonne explication intuitive, mais ne constitue pas une preuve sérieuse. C'est malheureusement assez classique en algorithmique. En second, le point crucial (« y sera accessible dans l'arborescence *à partir de x ») ne me semble pas justifié. Or c'est la proposition 5.18 qui permet de le justifier ; c'est le point clé des parcours en profondeur, et il n'est pas mentionné dans le BBC, ce qui est curieux.*

Pour un arc transverse $u \to v$, le fait que $r(u) > r(v)$ et que les intervalles $[r(u), fin(u)]$ et $[r(v), fin(v)]$ soient disjoints implique encore mieux : que $r(v) \leq fin(v) < r(u) \leq fin(u)$.

Tout ceci permet de classer les arcs comme montré en table 4.

Un *circuit* est un chemin $u \to \cdots \to u$ contenant au moins un arc. En particulier, les boucles sont des circuits. circuit

Lemme 4.7 (BBC, section 4.4.4, pages 108–109) *Le graphe G est sans circuit si et seulement si il n'a ni boucle ni arc arrière (par rapport au parcours en profondeur L).*

Démonstration. Si $u \to v$ est un arc arrière, alors $v \to_T^* u$ donc $v \to^* u$ (dans G), ce qui exhibe un circuit dans G.

Réciproquement, supposons qu'il n'y ait aucun arc arrière. Tous les arcs sont avant ou transverses. Ordonnons les sommets u par la forme suivante d'ordre lexicographique sur $(fin(u), r(u))$: $u \succ v$ si et seulement si $fin(u) > fin(v)$, ou bien $fin(u) = fin(v)$ et $r(u) < r(v)$. La table 4 montre que tout arc $u \to v$, qu'il soit avant ou transverse, satisfait $u \succ v$. Comme $\succ$ est un ordre strict (transitif, irréflexif), l'existence d'un circuit de u à u impliquerait $u \succ u$, ce qui est impossible. □

Le BBC ne mentionne pas les boucles : et pour cause, il ne traite que des graphes sans boucles.

5.6 Calcul des rangs et des temps de fin

On peut construire un parcours $L \stackrel{\text{def}}{=} [u_1, \cdots, u_n]$ de G en construisant ses préfixes L_k, par récurrence sur k, comme suit. D'abord, $L_1 \stackrel{\text{def}}{=} [*]$. (Par commodité, on posera aussi $L_0 \stackrel{\text{def}}{=} []$.) Ensuite, pour tout k $(1 \leq k \leq n-1)$, en supposant $L_k \stackrel{\text{def}}{=} [u_1, \cdots, u_k]$ construit, on note qu'il existe nécessairement un u_i $(1 \leq i \leq k)$ et un arc $u_i \to v$, avec $v \notin \{u_1, \cdots, u_k\}$. En effet, sinon, L_k serait une liste de sommets contenant $*$ $(= u_1)$ et stable par successeurs, et contiendrait donc Reach$(*)$. Or Reach$(*) = S$, donc le cardinal de L_k vaudrait au moins n, ce qui est impossible puisque $k \leq n-1$. On pose alors $L_{k+1} \stackrel{\text{def}}{=} [u_1, \cdots, u_k, v]$.

Si l'on souhaite prouve que $L = L_n$ est bien un parcours de G, il suffit de montrer les invariants :

```
fun dfs_1 (u) =
    (* Pre I_1(dfsNum, marked, r, fin) et r(u) = dfsNum + 1 *)
    marked := marked ∪ {u};
    r(u) := ++dfsNum;
    for each v ∈ succ(u) do
        if v ∉ marked
           then dfs_1 (v);
    fin(u) := dfsNum;
    (* Post I_1(dfsNum, marked, r, fin) et dfsNum = fin(u) = fin(u) *)
```

TABLE 5 – Un algorithme de parcours en profondeur basique

— chaque sommet de G apparaît au plus une fois dans L_k ;

— $*$ est le premier sommet de L_k ;

— chaque sommet de L_k, sauf le premier, est successeur d'un élément qui le précède dans L_k.

On ne demande donc pas que tout sommet de G apparaisse dans L_k. C'est la seule différence par rapport à un parcours.

Revenons sur la façon donnée ci-dessus de construire L_k. Nous choisissons maintenant i *maximal* parmi les indices possibles. L est alors un parcours en profondeur. Pour le prouver, on considère l'invariant supplémentaire :

— chaque sommet u_j de L_k, sauf le premier, est successeur d'un sommet ouvert de L_{j-1}.

(On laissera au lecteur le soin d'adapter la notion de sommet ouvert à ce cadre.) Si parmi les indices i possibles on avait choisi le plus petit, alors L aurait été un parcours en largeur.

Supposons maintenant G représenté par liste de successeurs. On a donc une fonction `succ`: $S \to S$ `list` qui à tout sommet associe une liste (ordonnée) de ses successeurs. Nous supposerons que cette liste est sans répétition.

On peut contraindre encore davantage la construction de L en demandant que, à i fixé, v soit le *premier* élément de $\texttt{succ}(u_i)$ qui ne soit pas dans $\{u_1, \cdots, u_k\}$. Le parcours en profondeur L est alors défini de façon unique.

L'algorithme de la table 5, ou plutôt l'appel `dfs_1 (*)`, calcule les rangs et les temps de fin de chaque sommet u d'un graphe G, avec source $*$. Le graphe est donné par une fonction `succ : 'a -> 'a list` donnant la liste des successeurs d'un sommet. Les variables `marked : 'a set` (ensemble des sommets déjà marqués), `dfsNum : int` (compteur maintenant le nombre de sommets déjà marqués), `r : 'a -> int ref` (fonction rang) et `fin : 'a -> int ref` (fonction temps de fin) sont supposées globales, pour simplifier. De plus, `dfsNum` est initialisée à 0, et `marked` à l'ensemble vide.

La pré-condition, repérée par `Pre`, est une hypothèse faite sur les entrées de `dfs_1`. Si elle est vraie, on prétend que la post-condition, repérée par `Post`, est vraie en sortie. Elles se lisent comme suit.

L'invariant $I_1(k, \texttt{marked}, \texttt{r}, \texttt{fin})$ est la conjunction des propriétés suivantes :

(i) `marked` est l'ensemble $\{u_1, \cdots, u_k\}$ des éléments de $L_k = [u_1, \cdots, u_k]$;

(ii) pour tout i $(1 \leq i \leq k)$, $\texttt{r}(u_i) = i$ $(= r(u_i))$;

(iii) pour tout sommet fermé u de L_k, $\texttt{fin}(u) = fin(u)$.

La condition $r(\texttt{u}) = \texttt{dfsNum}+1$, en conjonction avec l'invariant et la définition d'un parcours, implique que `u` n'est pas marqué (pas dans l'ensemble `marked`) en entrée.

La post-condition implique, elle, que `u` soit marqué, ainsi que tous ses descendants, dans $D_T(\texttt{u})$ — mais pas davantage : si k désigne la valeur de `dfsNum` en entrée et k' celle de `dfsNum` en sortie, alors $L_{k'}$ est exactement le préfixe de L obtenu en ajoutant à la fin de L_k les sommets de $D_T(\texttt{u})$.

À la fin du calcul de `dfs_1`$(*)$, la post-condition implique que `dfsNum` $= fin(*)$. Or $fin(*) = n$, et l'invariant implique donc que `r`, `fin` envoient chaque sommet de G vers son rang et son temps de fin, respectivement.

On laisse la preuve de correction de `dfs_1` en exercice. Pour l'effectuer, en plus des résultats déjà connus sur les parcours, on a besoin de la proposition 5.24 ci-dessous, qui permet d'assurer que `dfsNum` $= \texttt{fin}(\texttt{u}) = fin(\texttt{u})$ en post-condition. On remarque que ceci donne une définition par récurrence sur $n - r(u)$ de $fin(u)$ (voir la table 4), et c'est ce calcul par récurrence qui est réalisé par `dfs_1`, car les arcs de liaison $\texttt{u} \to_T \texttt{v}$ sont exactement ceux explorés lors de l'appel récursif `dfs_1 (v)`, c'est-à-dire lorsque $\texttt{v} \notin$ `marked`.

Lemme 5.23 *Pour tout sommet u de G, en posant $j \stackrel{def}{=} fin(u)$:*

1. *u_j est une feuille de l'arbre T ;*
2. *pour tout sommet v de G tel que $u \to_T v \to_T^* u_j$, $fin(v) = fin(u)$.*

Démonstration. 1. Si u_j n'est pas une feuille de T, il a un successeur v dans T. Or $u_j \to_T v$ implique $j = r(u_j) < r(v)$, ce qui est impossible, puisque $u \to_T^* u_j \to_V v$ implique que v est dans $D_T(u)$, donc que $r(v) \leq fin(u) = j$, par la proposition 5.19 (3).

2. Toujours par la proposition 5.19 (3), $u \to_T v$ implique $fin(v) \leq fin(u) = j$. Pour l'inégalité réciproque, par cette même proposition $v \to_T^* u_j$ implique $fin(u_j) \leq fin(v)$. Mais par le point 1, u_j est une feuille de T, donc $D_T(u_j) = \{u_j\}$, autrement dit $fin(u_j) = r(u_j)$ $(= j)$, par la proposition 5.18. Donc $j \leq fin(v)$. □

Proposition 5.24 *Pour tout sommet u de G, $fin(u)$ vaut :*

— *$r(u)$ si u est une feuille de T ;*

— $\max\{fin(v) \mid u \to_T v\}$ *sinon.*

Démonstration. Soit $fin'(u)$ l'entier $r(u)$ si u n'a pas de successeur dans T, $\max\{fin'(v) \mid u \to_T v\}$ sinon. On montre par récurrence (forte) sur $n - r(u)$ que $fin'(u) \leq fin(u)$. D'abord, on a toujours $r(u) \leq fin(u)$. Ensuite, si $u \to_T v$ alors $fin(v) \leq fin(u)$ par la proposition 5.19 (3), et par hypothèse de récurrence $fin'(v) \leq fin(v)$. En prenant le max sur tous les v tels que $u \to_T v$, on obtient $fin'(u) \leq fin(u)$.

On démontre que, réciproquement, $fin(u) \leq fin'(u)$ comme suit. Soit $i \stackrel{\text{def}}{=} r(u)$, $j \stackrel{\text{def}}{=} fin(u)$. Si u est une feuille de T, alors $D_T(u) = \{u\}$, donc $fin(u) = r(u)$, par la proposition 5.18, or $r(u) \leq fin'(u)$.

Sinon, $D_T(u) = \{u_i, u_{i+1}, \cdots, u_j\}$, et nécessairement $i < j$ (sinon $u = u_i$ serait une feuille de T). De plus, il existe un chemin de u à u_j dans T, et comme $i < j$, ce chemin est non vide, donc de la forme $u \to_T v \to_T^* u_j$. Par le lemme 5.23 (2), $fin(v) = fin(u)$, et par définition $fin(v) \leq fin'(u)$. □

En représentant chaque fonction `r`, `fin` par des tableaux indexés par les sommets, en supposant qu'ils soient identifiés par des numéros, et en codant l'ensemble `marked` par un tableau de bits, la complexité de `dfs_1` est $O(m + n)$, où n est le nombre de sommets de G et m son nombre d'arcs (somme pour chaque sommet `u` d'une constante, plus coût du parcours de `succ(u)`).

On peut ensuite décider de l'existence d'un circuit dans G en testant, dans une deuxième passe, s'il existe un arc arrière, en comparant pour chaque arc $u \to v$ les rangs et temps de fin de u et de v (table 4). Ceci prend un temps additionnel $O(m + n)$.

Plutôt que d'effectuer une deuxième passe, on peut aussi directement décider de l'existence d'un circuit au sein de l'algorithme `dfs_1`, en ajoutant une branche `else` (s'appliquant donc si `v` $\in$ `marked`) :

```
...else if u=v or (r(u)>r(v) and fin(u)≤fin(v)) then circuit :=
                                    true;
```

Pour ceci, on doit, avant de lancer `dfs_1`$(*)$, initialiser `fin`(v) à une valeur strictement plus grande que n, et `circuit` à `false`. L'invariant est raffiné en demandant que pour tout sommet ouvert u de L_k, `fin`$(u) = \infty$.

5.7 Tri topologique

Définition 5.25 *Une* extension $\sqsubseteq$ *d'une relation d'ordre* $\leq$ *sur un ensemble* E *est une relation d'ordre telle que pour tous* $x, y \in E$*, si* $x \leq y$ *alors* $x \sqsubseteq y$ *— autrement dit,* $\leq \subseteq \sqsubseteq$. extension

Une extension totale *est une extension qui est une relation d'ordre totale.* extension totale

Le théorème de Szpilrajn énonce que tout ordre a une extension totale. En voici une démonstration rapide, et totalement non constructive. La famille des

extensions de $\leq$ est un ensemble ordonné inductif, et a donc un élément maximal par le lemme de Zorn. Si $\sqsubseteq$ est une extension de $\leq$ mais n'est pas totale, il existe deux éléments x et y tels que $y \not\sqsubseteq x$. La relation $\sqsubseteq'$ définie par $a \sqsubseteq' b$ ssi $a \sqsubseteq b$, ou bien $a \sqsubseteq x$ et $y \sqsubseteq b$, est alors une extension stricte de $\sqsubseteq$. Ceci montre que toute extension non totale est non maximale. Donc l'extension maximale trouvée par le lemme de Zorn est totale.

Dans le cas où E est un ensemble fini, une extension totale $\sqsubseteq$ de $\leq$ est souvent appelée un tri topologique. Ceci s'applique notamment à l'ensemble des sommets d'un graphe G sans circuit, ordonnés par $\rightarrow^*$. (On a besoin que G soit sans circuit, ou au moins que ses seuls circuits soient des boucles, pour que $\rightarrow^*$ soit antisymétrique.) Un tel tri topologique peut se présenter sous la forme d'une énumération $x_1 \sqsubset x_2 \sqsubset \cdots \sqsubset x_n$ des éléments de E, donc d'une numérotation des éléments de E : le numéro d'un élément x est l'unique i tel que $x = x_i$.

Définition 5.26 *Un* tri topologique *d'un graphe $G \stackrel{def}{=} (S, A)$ est une fonction de numérotation $to\colon S \to \mathbb{N}$ telle que pour tous $u, v \in S$, si $u \to v$ alors $to(u) < to(v)$.* tri topologique

Un tri topologique inverse *est une fonction de numérotation $rto\colon S \to \mathbb{N}$ telle que pour tous $u, v \in S$, si $u \to v$ alors $rto(u) > rto(v)$.* tri topologique inverse

On définit aussi un tri topologique inverse, car c'est ce que l'on va calculer le plus naturellement. On peut passer de l'un à l'autre par $rto(u) \stackrel{\text{def}}{=} N - to(u)$, où N est un majorant des valeurs prises par to, et réciproquement.

Remarque 5.27 (Rang≠tri topologique) *Le parcours $L \stackrel{def}{=} [u_1, \cdots, u_n]$ en profondeur d'un graphe G avec une source $*$ donne une fonction de numérotation r (la fonction rang), et l'on pourrait penser que c'est un tri topologique $u_1 \sqsubset \cdots \sqsubset u_n$. Ceci est faux. Comme la table 4 le montre, ceci échoue dès qu'il existe un arc transverse.*

Nous allons démontrer qu'un graphe a un tri topologique si et seulement si c'est un DAG, c'est-à-dire si et seulement s'il est sans circuit. On peut démontrer la direction difficile (si), par exemple, en utilisant le théorème de Szpilrajn, appliqué à la relation $\rightarrow^*$, qui est une relation d'ordre dès que G est un DAG. Nous allons en donner une démonstration plus algorithmique.

On observe d'abord qu'on peut se ramener au cas d'un graphe avec une source.

Lemme 5.28 *Soit G un graphe. Alors :*

1. *G est sans circuit si et seulement si G_* est sans circuit ;*
2. *si G a un tri topologique inverse rto, alors G_* a un tri topologique inverse rto_*, qui coïncide avec rto sur les sommets de G ;*
3. *si G_* a un tri topologique inverse, alors sa restriction aux sommets de G est un tri topologique inverse de G.*

Démonstration. 1. Clairement, tout circuit de G est un circuit dans G_*. Si $x_0 \to x_1 \to \cdots \to x_k$ ($k \geq 1$) est un circuit dans G_*, nous montrons qu'il est nécessairement un circuit de G. Sinon, l'un des x_i est égal à $*$. Comme $*$ n'a pas d'arc entrant, nécessairement $k = 0$. Mais $x_0 = x_k$, et l'arc entrant $x_{k-1} \to x_k$ est lui aussi impossible.

2. On pose $rto_*(u) \stackrel{\text{def}}{=} rto(u)$ pour tout sommet u de G, et $rto_*(*)$ égal à un entier quelconque strictement supérieur à $rto(u)$ pour tout sommet u de G. Pour tout arc $u \to v$ de G, on a $rto(u) > rto(v)$, donc $rto_*(u) > rto_*(v)$. Les autres arcs de G_* sont de la forme $* \to u$, où u est un sommet de G, et alors $rto_*(*) > rto_*(u)$ ($= rto(u)$) est par construction.

3. Évident.

Lemme 5.29 *Soit G un graphe avec une source $*$, L un parcours en profondeur de G, et T l'arbre couvrant associé. Soit $rto\colon S \to \mathbb{N}$ n'importe quelle fonction telle que, pour tout sommet u,*

1. *pour tout $v \in S$ tel que $u \to_T v$, $rto(u) > rto(v)$;*
2. *pour tout $v \in S$ tel que $fin(v) < r(u)$, $rto(u) > rto(v)$.*

Si G est sans circuit, alors pour tous $u, v \in S$, $u \to v$ implique $rto(u) > rto(v)$.

Démonstration. Il n'y a que des arcs avant et des arcs transverses par le lemme 4.7 (BBC).

Si $u \to v$ est un arc avant, alors $v \in D_T(u)$, et comme G n'a pas de boucle, on a $v \neq u$, donc $u \to_T^+ v$. Par la condition 1, itérée autant de fois qu'il y a d'arcs de liaison de u à v, $rto(u) > rto(v)$.

Si $u \to v$ est un arc transverse, alors $fin(v) < fin(u)$. De plus, comme u et v sont incomparables pour $\to_T^*$, la proposition 5.20 nous indique que les intervalles $[r(v), fin(v)]$ et $[r(u), fin(u)]$ sont disjoints. Donc $fin(v) < r(u)$. On en déduit que $rto(u) > rto(v)$, par la condition 2. □

Toujours en supposant G sans circuit, on peut définir une telle fonction rto par :

$$rto(u) \stackrel{\text{def}}{=} \max(\, \max\{rto(v) \mid v \in S \text{ tel que } u \to_T v\}, \quad \max\{rto(v)) \mid v \in S \text{ tel que } fin(v) < r(u)\}) + 1, \tag{1}$$

où, le cas échéant, le max d'une famille vide est considéré égal à 0. Cette définition est une définition par récurrence sur $(fin(u), r(u))$ ordonné par le produit lexicographique de $>$ et de $<$, c'est-à-dire dans l'ordre strict $\succ$ introduit dans la démonstration du lemme 4.7 (BBC). Informatiquement parlant, les appels « récursifs » à $rto(v)$ sur le côté droit sont tous tels que $fin(u) > fin(v)$, ou bien $fin(u) = fin(v)$ et $r(u) < r(v)$.

Théorème 5.30 *Un graphe G a un tri topologique (inverse) si et seulement si G est un DAG.*

```
fun dfs_2 (u) =
    (* Pre I_2(dfsNum, marked, r, fin, rto, revTopOrder) et r(u) = dfsNum + 1 *)
    marked := marked ∪ {u};
    r(u) := ++dfsNum;
    for each v ∈ succ(u) do
        if v ∉ marked
           then dfs_2 (v);
    fin(u) := dfsNum;
    rto(u) := ++revToporder; (* calcul de rto(u) *)
    (* Post I_2(dfsNum, marked, r, fin, rto, revTopOrder) et dfsNum = fin(u) = fin(u) *)
```

TABLE 6 – Tri topologique

Démonstration. On peut se ramener au cas où G a une source $*$ par le lemme 5.28. Nous avons démontré la direction « si ». Réciproquement, si G a un tri topologique inverse rto, alors il ne peut pas avoir de circuit $x_0 \to x_1 \to \cdots \to x_k = x_0$ ($k \geq 1$). En effet, sinon $rto(x_0) > rto(x_1) > \cdots > rto(x_k) = rto(x_0)$, ce qui est impossible car $k \geq 1$. □

L'algorithme de la table 6, qui est obtenu à partir de `dfs_1` en ajoutant une seule ligne, étiquetée « calcul de rto(u) », et calcule $rto(u)$, pour chaque sommet u, en le stockant dans le tableau `rto`. La variable globale `revToporder` est initialisée à 0. L'invariant I_2(`dfsNum`, `marked`, `r`, `fin`, `rto`) est la conjunction de I_1(`dfsNum`, `marked`, `r`, `fin`) et de :

(iv) pour tout sommet fermé v de L_k, `rto`$(v) = rto(v)$,

(v) `revTopOrder` $= \max\{rto(v) \mid v$ sommet fermé de $L_k\}$,

où rto est définie en (1). (On rappelle que le max d'un ensemble vide vaut 0 par convention.)

Pour la même raison que pour `dfs_1`, `dfs_2 (*)` est un algorithme en temps $O(m+n)$, où n est le nombre de sommets de G, et m son nombre d'arcs.

Cet algorithme ne fonctionne que si G a une source $*$. Sinon, on peut soit la créer, soit simuler le déroulement de l'algorithme sur G_*. Il suffit d'écrire :

```
marked := ∅; (* au lieu de {*} *)
dfsNum := 1; (* au lieu de dfsNum := 0; r(*) := ++dfsNum *)
revTopOrder := 0;
for each v ∈ S do
    if v ∉ marked
       then dfs_2 (v);
```

6 Composantes fortement connexes

La relation d'accessibilité $\rightarrow^*$ d'un graphe orienté $G \stackrel{\text{def}}{=} (S, A)$ est un préordre (une relation réflexive et transitive). Tout préordre a une relation d'équivalence associée, dans ce cas la relation $\equiv$ définie par $u \equiv v$ si et seulement si $u \rightarrow^* v$ et $v \rightarrow^* u$.

Définition 6.1 *Les* composantes fortement connexes *d'un graphe orienté sont ses classes d'équivalence pour la relation d'équivalence $\equiv$ associée à sa relation d'accessibilité $\rightarrow^*$. On notera $[u]$ la classe d'équivalence de u pour cette relation d'équivalence, et on l'appellera la* composante fortement connexe de u.

composantes fortement connexes

composante fortement connexe de u

Remarque 6.2 *Il ne faut surtout pas dire « composante connexe » pour un graphe orienté. On appelle en général composante connexe d'un graphe orienté G tout composante connexe du graphe non orienté sous-jacent, défini comme ayant une arête u — v pour chaque arc $u \rightarrow v$ de G. On a alors que $u \equiv v$ implique u —* v, mais la réciproque est fausse : une composante connexe est en général une* union *de composantes fortement connexes de G.*

On peut former le quotient $S/\equiv$, dont les éléments sont les composantes fortement connexes de G. De plus, on peut définir une relation $\Rightarrow$ sur $S/\equiv$ par : $C \Rightarrow C'$ si et seulement si $C \neq C'$ et il existe $u \in C$ et $v \in C'$ tels que $u \rightarrow v$ soit un arc de G. Le graphe $(S/\equiv, \Rightarrow)$ est alors sans circuit.

Définition 6.3 *Pour tout graphe orienté $G \stackrel{def}{=} (S, A)$, le graphe $(S/\equiv, \Rightarrow)$ est la* condensation *de G.*

condensation

Remarque 6.4 *Dans un circuit $u_0 \rightarrow u_1 \rightarrow \cdots \rightarrow u_n = u_0$ ($n \geq 1$), tous les sommets appartiennent à la même composante fortement connexe. En effet, pour tous indices i et j ($0 \leq i, j \leq n$), disons $i \leq j$, on a $u_i \rightarrow^* u_j$, mais aussi $u_j \rightarrow^* u_n = u_0 \rightarrow^* u_i$. Donc $u_i \equiv u_j$.*

Lorsque G est un graphe qui n'a pas forcément de source, on note que $\{*\}$ forme une composante fortement connexe dans le graphe G_* à elle seule. Les composantes fortement connexes de G_* sont donc celles de G, plus $\{*\}$. On peut donc, de nouveau, supposer que G a une source $*$, sans perdre de généralité, si l'on souhaite calculer ses composantes fortement connexes.

6.1 Points d'entrée, structure des composantes fortement connexes

Définition 6.5 *Soit $G \stackrel{def}{=} (S, A)$ un graphe avec une source $*$, soit L un parcours en profondeur de G, et soit $T \stackrel{def}{=} (S, *, p)$ l'arbre couvrant associé. Pour toute composante fortement connexe C de G, le* point d'entrée *$*_C$ de C est le sommet de plus petit rang de C.*

point d'entrée

Lemme 6.6 *Pour toute composante fortement connexe C de G, $C \subseteq D_T(*_C)$.*

Démonstration. Soit $i \stackrel{\text{def}}{=} r(*_C)$, $j \stackrel{\text{def}}{=} fin(*_C)$. Par la proposition 5.18, les sommets de C sont exactement ceux de rangs compris entre i et j.

On le démontre par récurrence sur la longueur d'un chemin de $*_C$ à v. Si cette longueur vaut 0, $v = *_C$ est évidemment dans $D_T(*_C)$. Sinon, ce chemin est de la forme $*_C \to^* u \to v$, où le chemin $*_C \to^* u$ est strictement plus court. De plus, comme $v \equiv *_C$, il existe un chemin de v à $*_C$ dans G, dont aussi un chemin $u \to v \to^* *_C$: comme $*_C \to^* u$ et $u \to^* *_C$, u est donc aussi dans C. On peut dès lors appliquer l'hypothèse de récurrence, et en déduire que u est dans $D_T(*_C)$. On regarde maintenant les différentes formes possibles de l'arc $u \to v$:

- boucle : alors $v = u$, et donc trivialement $v \in D_T(*_C)$;
- arc avant : alors $u \to^*_T v$, et comme $*_C \to^*_T u$ (puisque $u \in D_T(*_C)$), on a $*_C \to^*_T v$ donc $v \in D_T(*_C)$;
- reste le cas où $u \to v$ est un arc arrière ou transverse, qui est l'unique cas intéressant. On a alors $r(v) < r(u)$ (voir la table 4). On utilise maintenant le fait que, par la proposition 5.18, les sommets de C sont exactement ceux de rangs compris entre $i \stackrel{\text{def}}{=} r(*_C)$ et $j \stackrel{\text{def}}{=} fin(*_C)$. Comme $u \in D_T(*_C)$, on a donc $i \leq r(u) \leq j$. Comme $v \in C$ et $*_C$ soit de rang minimal dans C, on a $i \leq r(v)$. Donc $i \leq r(v) \leq j$, ce qui signifie que v est dans $D_T(*_C)$. □

Lemme 4.8 (BBC, section 4.4.4, page 109, reformulé) *Pour toute composante fortement connexe C de G, $T_{|C} \stackrel{def}{=} (C, *_C, p_{|C \setminus \{*_C\}})$ est un arbre couvrant C, de racine $*_C$, et un sous-graphe induit de T.*

Ce qui est important, c'est que ceci veut dire qu'il y a *exactement un* point d'entrée de T dans C, qui est $*_C$.

Démonstration. La seule chose qui ne soit pas une évidence est que, pour tout sommet $v \in C \setminus \{*_C\}$, $p(v)$ est encore dans C. Par le lemme 6.6, v est dans $D_T(*_C)$, autrement dit il y a chemin $*_C \to^*_T v$ (dans T, pas juste dans G). Ce chemin est unique, et comme $v \neq *_C$, il est de la forme $*_C \to^*_T p(v) \to_T v$.

Comme v et $*_C$ sont dans la même composante fortement connexe, on a aussi un chemin $v \to^* *_C$, donc un circuit $*_C \to^*_T p(v) \to_T v \to^* *_C$. Par la remarque 6.4, tous les sommets de ce circuit sont dans C, en particulier $p(v)$. □

6.2 Points d'attache

Pour tout sommet u d'une composante fortement connexe C, le lemme 4.8 (BBC) montre que u est un sommet de l'arbre $T_{|C}$, et donc il existe un (unique) chemin $*_C \to^*_T u$. L'algorithme de Tarjan (R. E. Tarjan, *Depth-First Search and Linear Graph Algorithms*, SIAM Journal on Computing, 1(2) :146–160, 1972) va calculer, pour chaque sommet u, un indice $\texttt{low}(u)$ qui est l'indice d'un sommet

qu'on appelle son point d'attache $a(u)$. On pourrait penser que $a(u)$ est justement $*_C$, mais ce ne sera pas le cas en général ! En revanche, on verra que $u = a(u)$ si et seulement si $u = *_C$.

La définition du point d'attache est technique. Les circuits sans boucle partant de et arrivant en u (de composante fortement connexe $C \stackrel{\text{def}}{=} [u]$) sont tous de la forme $u \to_T^* v \hookrightarrow w \to^* u$, où $v \hookrightarrow w$ est le premier arc en arrière ou transverse du circuit. (On notera $\hookrightarrow$, en général, pour désigner un arc arrière ou transverse.) Il existe nécessairement un tel arc dans chacun de ces circuits : sinon on aurait $u \to_T^+ u$, et donc $r(u) < r(u)$, ce qui est absurde.

Lorsque l'on parcourt l'espace de tous les circuits de la forme $u \to_T^* v \hookrightarrow w \to^* u$ (avec ou sans boucle, peu importe), les sommets w ainsi obtenus forment un ensemble $AT(u)$.

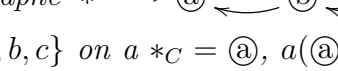
Définition 6.7 *Soit G un graphe avec une source $*$. On note $v \hookrightarrow w$ pour un arc arrière ou transverse de v à w.*

Le point d'attache *$a(u)$ d'un sommet u est le sommet $w \in AT(u) \cup \{u\}$ de plus petit rang.* point d'attache

Les sommets de $AT(u)$ sont les sommets w tels qu'il existe un circuit de la forme $u \to_T^* v \hookrightarrow w \to^* u$, pour un certain sommet v.

On note $low(u)$ le rang $r(a(u))$ du point d'attache de u.

Le point d'attache est donc le sommet de $AT(u)$ de plus petit rang s'il existe un $w \in AT(u)$ de rang inférieur à celui de u, sinon c'est u lui-même.

Remarque 6.8 *La figure 4.8 du BBC peut donner l'impression que $a(u)$ est égal à $*_C$ pour tout sommet u de toute composante fortement connexe C. Ce n'est pas le cas. Par exemple, dans le graphe*  *, dans la composante fortement connexe $C \stackrel{def}{=} \{a, b, c\}$ on a $*_C = ⓐ$, $a(ⓐ) = a(ⓑ) = ⓐ$, mais $a(ⓒ) = ⓑ$.*

Il vaut la peine de remarquer les propriétés suivantes du point d'attache $a(u)$ et de son rang $low(u)$.

Remarque 6.9

1. *$low(u) \leq r(u)$;*
2. *si $u \to_T^* v \hookrightarrow w \to^* u$, alors $low(u) \leq r(w)$;*
3. *$low(u) < r(u)$ si et seulement s'il existe un circuit de la forme $u \to_T^* v \hookrightarrow w \to^* u$ où $r(w) < r(u)$.*

Lemme 6.10 *Pour toute composante fortement connexe C, pour tout sommet u de C, $a(u)$ est aussi dans C.*

Démonstration. C'est clair si $a(u) = u$. Sinon, $a(u)$ est un sommet w apparaissant dans un circuit $u \to_T^* v \hookrightarrow w \to^* u$, lequel ne contient que des sommets de C par la remarque 6.4. □

Lemme 4.9 (BBC, section 4.4.4, page 110) *Un sommet u est le point d'entrée $*_C$ de sa composante fortement connexe $C \stackrel{def}{=} [u]$ si et seulement si $u = a(u)$.*

Démonstration. Soit C la composante fortement connexe de u, et $i \stackrel{\text{def}}{=} r(u)$. Si $u = *_C$, alors tout circuit de la forme $u \rightarrow_T^* v \hookrightarrow w \rightarrow^* u$ ne contient que des sommets de C, par la remarque 6.4. Comme w est dans C, et que $*_C$ est le sommet de plus petit rang dans C par définition, $i \leq r(w)$. Ceci étant vrai pour tous les sommets w ainsi obtenus, il s'ensuit que $a(u) = u$.

Réciproquement, supposons que $u = a(u)$, c'est-à-dire que dans tout circuit de la forme $u \rightarrow_T^* v \hookrightarrow w \rightarrow^* u$, on a $r(w) \geq i$. On va montrer que $u = *_C$.

En particulier, pour tout chemin de la forme $u \rightarrow_T^* v \hookrightarrow w \rightarrow^* *_C$, on a $r(w) \geq i$. En effet, un tel chemin induit un circuit $u \rightarrow_T^* v \hookrightarrow w \rightarrow^* *_C \rightarrow^* u$, puisque u est $*_C$ sont dans la même composante fortement connexe.

Comme u est dans C, il existe un chemin $u \rightarrow^* *_C$. Choisissons-en contenant le nombre minimal d'arcs arrières ou transverses. Si ce chemin $u \rightarrow^* *_C$ contient au moins un arc arrière ou transverse, alors on peut l'écrire $u \rightarrow_T^* v \hookrightarrow w \rightarrow^* *_C$. Dans ce cas, on a vu que $r(w) \geq i$. Comme $v \hookrightarrow w$ est arrière ou transverse, $r(w) < r(v)$ (voir la table 4). Par la proposition 5.18, les descendants dans T du sommet u sont exactement les sommets de rangs compris entre $r(u)$ et $fin(u)$; comme $u \rightarrow_T^* v$, on a donc $r(v) \leq fin(u)$. On en déduit que $r(w) < r(v) \leq fin(u)$; donc $i = r(u) \leq r(w) \leq fin(u)$, et donc, par la proposition 5.18 encore, $u \rightarrow_T^* w$. Mais alors $u \rightarrow_T^* w \rightarrow^* *_C$ est un chemin contenant un arc arrière ou transverse de moins, ce qui contredit la minimalité.

Il s'ensuit que le chemin $u \rightarrow^* *_C$ ne contient aucun arc arrière ou transverse. On en élimine les boucles, et il reste un chemin ne contenant que des arcs avant. Donc $u \rightarrow_T^* *_C$.

L'unique sommet de $T_{|C}$ dont $*_C$ est un descendant est $*_C$. Donc $u = *_C$. □

La proposition 6.11 ci-dessous fournit une définition équivalente de $low(u)$ (et donc de $a(u)$) par récurrence sur $n - r(u)$. En effet, pour tout arc avant $u \rightarrow v$, on a $r(u) < r(v)$ (voir la table 4).

Proposition 6.11 *Pour tout sommet u de G, le rang de $a(u)$ est l'entier le plus petit parmi :*

1. *$r(u)$;*
2. *$r(a(v))$ (c'est-à-dire $low(v)$), lorsque $u \rightarrow v$ parcourt les arcs de liaison d'origine u (autrement dit, $u \rightarrow_T v$) ;*
3. *$r(v)$, lorsque $u \rightarrow v$ parcourt les arcs arrières ou transverses d'origine u dont l'extrémité v est dans la même composante fortement connexe que u.*

Démonstration. Notons que dans le cas 2, on ne restreint pas v à être dans la même composante fortement connexe que u. Mais seuls les v qui sont dans la même composante fortement connexe que u comptent, car : $(*)$ si $u \rightarrow_T v$

et si u et v ne sont pas dans la même composante fortement connexe, alors $r(a(u)) < r(a(v))$. En effet, soit C la composante fortement connexe celle de v. Comme $u = p(v)$ n'est pas dans C, c'est que $v = *_C$, par le lemme 4.8 (BBC). Donc $r(u) < r(v) = r(*_C)$ (voir la table 4). Or, comme le point d'entrée $*_C$ est le sommet de rang minimal de C', et que $a(v)$ est dans C, on a $r(*_C) \leq r(a(v))$. Donc $r(u) < r(a(v))$. Or, par définition, $r(a(u)) \leq r(u)$, ce qui établit $(*)$.

Montrons le lemme. Soit C la composante fortement connexe de u.

Commençons par montrer que $r(a(u))$ est inférieur ou égal aux entiers mentionnés dans les cas 1, 2, et 3.

— Cas 1 : $r(a(u)) \leq r(u)$ est par définition du point d'attache.

— Cas 2. Soit $u \to_T v$ un arc de liaison. On prétend que $r(a(u)) \leq r(a(v))$. Si u et v ne sont pas dans la même composante fortement connexe, alors $r(a(u)) < r(a(v))$ par $(*)$. Sinon, $v \in C$, et l'on va construire un circuit d'origine et d'extrémité u, passant par v et $a(v)$, comme suit. D'abord, $r(a(v)) < r(v)$, sinon $a(v) = v$, ce qui impliquerait $v = *_C$, par le lemme 4.9 (BBC) ; mais $*_C$ étant la racine du sous-arbre $T_{|C}$, par le lemme 4.8 (BBC), $u \to_T v$ impliquerait que u ne serait pas dans C. Donc $r(a(v)) < r(v)$, et ceci implique qu'il existe un circuit $v \to_T^* v' \hookrightarrow a(v) \to^* v$, par définition du point d'attache. On peut alors former le circuit $u \to_T v \to_T^* v' \hookrightarrow a(v) \to^* v \to^* u$, où la dernière partie $v \to^* u$ vient du fait que u et v sont dans la même composante fortement connexe, C. Par la propriété de minimalité dans la définition du point d'attache de u, on a donc $r(a(u)) \leq r(a(v))$.

— Cas 3. Soit $u \to v$ un arc arrière ou transverse, avec $v \in C$. On prétend que $r(a(u)) \leq r(v)$. Comme $v \in C$, on a $v \to^* u$, ce qui nous permet de former un circuit $u \to v \to^* u$. Ceci est un circuit $u \to_T^* u \hookrightarrow v \to^* u$ (avec 0 arc de u à u), et la minimalité dans la définition du point d'attache de u implique donc que $r(a(u)) \leq r(v)$.

Il reste à montrer que $r(a(u))$ est supérieur ou égal à l'un des entiers des cas 1, 2, ou 3. Il sera donc égal au plus petit de ces entiers.

— Si $a(u) = u$ (c'est-à-dire si $u = *_C$), alors $r(a(u)) = r(u)$ est égal à l'entier du cas 1. Dans la suite, supposons que $a(u) \neq u$, donc $a(u)$ est un sommet w tel que $r(w) < r(u)$, et il existe un circuit de la forme $u \to_T^* v' \hookrightarrow w \to^* u$.

— Si $u = v'$, d'abord, par la remarque 6.4, v' est dans C. On pose $v \stackrel{\text{def}}{=} w$: c'est un sommet du cas 3. Et $r(a(u)) = r(w) \geq r(v)$, trivialement.

— Sinon, le chemin de u à v' est non vide, et notre circuit s'écrit donc $u \to_T v \to_T^* v' \hookrightarrow w \to^* u$. Le sommet v est un sommet du cas 2. Il ne reste qu'à montrer que $r(a(u)) = r(w)$ est supérieur ou égal à $r(a(v))$. Pour ceci, on fait « tourner le circuit d'un cran », et l'on produit le circuit $v \to_T^* v' \hookrightarrow w \to^* u \to_T v$. La minimalité dans la définition du point d'attache $a(v)$ de v implique alors que $r(a(v)) \leq r(w)$. □

6.3 Caractérisations récursives des points d'attache

La proposition 6.11 montre comment calculer $a(u)$, récursivement... à condition de savoir déterminer si u et v sont dans la même composante fortement connexe dans le cas 3. Ceci sera effectué dans l'algorithme de Tarjan en testant si v est sur une pile gardée dans une variable nommée `stk`. En fait, comme on le verra à la proposition 6.13, les sommets de cette pile sont ceux qui sont dans la même composante fortement connexe que v, et aussi ceux de $D_T(u)$. On pourrait exclure ce dernier cas en remplaçant le test '`v` $\in$ `stk`' de la ligne 9 de l'algorithme à venir (table 7) par '`v` $\in$ `stk and r(v)<r(u)`'. Mais ce test est inutile, grâce à l'affaiblissement suivant de la proposition 6.11.

Proposition 6.12 *Pour tout sommet u de G, le rang de $a(u)$ est l'entier le plus petit parmi :*

1. *$r(u)$;*
2. *$r(a(v))$, lorsque $u \to v$ parcourt les arcs de liaison d'origine u (autrement dit, $u \to_T v$) ;*
3. *$r(v)$, lorsque $u \to v$ parcourt :*
   (a) *les arcs arrières ou transverses d'origine u dont l'extrémité v est dans la même composante fortement connexe que u ;*
   (b) *et les arcs avants, ainsi que les boucles, d'origine u.*

Démonstration. Il suffit de montrer que les arcs avants (ou les boucles) d'origine u ne contribuent pour rien dans le minimum de la proposition 6.11. Il suffit pour cela d'observer que pour tout arc avant (ou boucle) $u \to v$, on a $u \to_T^* v$ donc $r(u) \leq r(v)$. Dès lors, les valeur $r(v)$ du cas 3(b) sont toutes plus grandes que la valeur du cas 1. □

Rappelons que nous supposons un parcours en profondeur $L \stackrel{\text{def}}{=} [u_1, \cdots, u_n]$. Pour tout sommet u du graphe G, on peut regarder la suite des sommets u, $p(u)$, ..., $p^i(u) = *$. Prise à l'envers, c'est l'unique chemin de $*$ à u dans T : $* = u_{i_1} \to_T u_{i_2} \to_T \cdots \to_T u_{i_\ell} = u$, avec $1 = i_1 < i_2 < \cdots < i_\ell = k$. On appellera ce chemin la *branche* menant au sommet u. Par définition, l'un des sommets u_{i_j} est le point d'entrée $*_C$ de la composante fortement connexe de u.

branche

Pour tout k ($0 \leq k \leq n$), notons π_k la sous-liste de L_k formée des éléments u_i qui sont dans la même composante fortement connexe qu'un des éléments de la branche $* = u_{i_1} \to_T u_{i_2} \to_T \cdots \to_T u_{i_\ell} = u_k$ menant à u_k. Ce n'est pas l'union des composantes fortement connexes de u_{i_1}, u_{i_2}, ..., u_{i_ℓ}, mais c'est l'intersection de cette union avec l'ensemble des éléments de L_k, listée dans le même ordre que dans L_k.

On appellera π_k la *pile à profondeur k*. Ce sera ce qui sera stocké dans la pile `stk` de l'algorithme de Tarjan plus bas.

pile à profondeur k

Proposition 6.13 *Soit u un sommet de G, et $k \in [r(u), fin(u)]$. Pour tout arc $u \to v$ de source u :*

1. *si $u \equiv v$ et $v \in L_k$, alors v est dans π_k ;*
2. *si v est dans π_k, alors v est dans L_k, et $u \equiv v$ ou bien $u \to_T^* v$.*

Démonstration. Notons $* = u_{i_1} \to_T u_{i_2} \to_T \cdots \to_T u_{i_\ell} = u_k$ la branche menant à u_k (donc $i_\ell = k$). Par la proposition 5.18, et comme $k \in [r(u), fin(u)]$, u_k est dans l'ensemble $D_T(u)$ des descendants de u dans T. Donc $u = p^m(u_k)$ pour un certain entier $m \in \mathbb{N}$, d'où l'on déduit que $r(u) = i_{\ell-m}$. En d'autres termes, $u = u_{i_j}$ pour un certain entier j, $1 \leq j \leq \ell$.

1. Si $u \equiv v$, alors v est dans la même composante fortement connexe que u_{i_j}, et comme il est dans L_k, il est donc dans π_k.

2. Supposons maintenant que v est dans π_k. Alors $v \equiv u_{i_{j'}}$ pour un certain j', $1 \leq j' \leq \ell$. Choisissons j' minimal. En notant C la composante fortement connexe de v, $u_{i_{j'}}$ est alors la racine $*_C$ de l'arbre $T_{|C}$ couvrant C, par le lemme 4.8 (BBC). En effet, sinon $p(u_{i_{j'}}) = u_{i_{j'-1}}$ serait aussi dans $T_{|C}$, donc dans C, ce qui contredirait la minimalité de j'.

Si $j' \geq j$, on a un chemin $u = u_{i_j} \to_T^+ u_{i_{j'}} = *_C \to_T^* v$ (puisque v est dans C, donc apparaît comme un sommet de T_C).

Si $j' < j$, alors on a $v \to^* u_{i_{j'}}$ (puisque $v \equiv u_{i_{j'}}$) $\to_T^* u_{i_j} = u$. Or, comme v est un successeur de u dans G, on a aussi $u \to v$, donc $u \equiv v$. □

6.4 Calcul des composantes fortement connexes

algorithme de Tarjan

L'*algorithme de Tarjan* du calcul des composantes fortement connexes de G (toujours supposé avec origine $*$) consiste à appeler `scc (*)`, où `scc` est défini en table 7. Il remplit un tableau `c`. À la fin du calcul, pour tout sommet u de G, `c`(u) vaudra le point d'entrée $*_C$ de sa composante fortement connexe C. Ceci permettra de décider ensuite très rapidement si deux sommets u et v sont dans la même composante fortement connexe : ils suffira de tester si `c`$(u) =$ `c`(v).

La procédure `scc` est une modification de `dfs_1`, où :

— d'abord, on ne calcule plus les temps de fin (mais on pourrait), et surtout,

— on calcule un nouveau tableau `low`, qui à chaque sommet va associer le rang dans L de son point d'attache ;

— on maintient une pile `stk`, sur laquelle on empile `u` (ligne 4) ; de façon cruciale, on ne dépile pas `u` à la fin de la boucle des lignes 5–9 : le but est d'accumuler tous les sommets de la composante fortement connexe [`u`] de `u` à la suite de `u` dans `stk` ;

— on ne dépile `stk` qu'aux lignes 10–11, lorsqu'on peut garantir que *tous* les sommets de [`u`] ont été empilés à la suite de `u` dans `stk` ;

```
   fun scc (u) =
   (* Pre I3(dfsNum, marked, r, stk, low) et r(u) = dfsNum + 1 et Pre(stk, u) *)
1        marked := marked ∪ {u};
2        r(u) := ++dfsNum;
3        low(u) := dfsNum;
4        push(stk,u);
5        for each v ∈ succ(u) do
6            if v ∉ marked
7               then (scc (v);
8                     low(u) := min(low(u),low(v)));
9            else if v ∈ stk then low(u) := min(low(u),r(v));
10       if r(u)=low(u) (* composante trouvée *)
11          then do v := pop(stk); c(v):=u until v=u; (* on dépile la composante *)
   (* Post I3(dfsNum, marked, r, stk, low) et dfsNum = fin(u) et Post(stk, u) *)
```

TABLE 7 – L'algorithme de Tarjan

— on aura besoin de tester rapidement l'appartenance d'un sommet à la pile `stk` en ligne 9 (non, on n'effectuera pas un parcours linéaire des éléments de `stk` ; voir plus bas).

La pile `stk` peut être implémentée par :

— un tableau `s` de sommets, de taille n ;

— un compteur `k` du nombre d'éléments dans `s` ;

— un tableau `onStk` de bits, de taille n ;

Ce dernier tableau sera tel que `onStk`$[u]$ sera vrai, pour tout sommet u, si et seulement si u est sur la pile. La pile est initialisée à `k` := `0`, et toutes les entrées de `onStk` sont mises à 0. Ceci prend un temps $O(n)$. L'empilement `push(stk,u)` s'implémente par : `s[k++] := u; on[u]:=true`. Le dépilement `pop(stk)` s'implémente par `let v = s[--k] in (on[v] := false; v)`. Le test `v` $\in$ `stk` s'implémente par `onStk[v]`. Ces trois opérations sont en temps constant.

Une pile `stk` dénote une liste de sommets P, la pile vide dénotant la liste vide $[]$, et le résultat de l'empilement de u sur une pile dénotée par $P \stackrel{\text{def}}{=} [u_1, \cdots, u_i]$ étant $P :: u \stackrel{\text{def}}{=} [u_1, \cdots, u_i, u]$. Par I_3(`dfsNum`, `marked`, `r`, `stk`, `low`), on entend l'invariant conjonction des propriétés **(i)** et **(ii)** utilisées pour définir I_1 (la condition **(iii)** n'a plus de sens, puisque le tableau `fin` n'est plus présent ici), et de :

(vi) pour tout sommet $u \in$ `marked`, `low`(u) est l'entier le plus petit parmi :

1. $r(u)$;
2. `low`(v), lorsque $u \rightarrow_T v$ parcourt les arcs de liaison d'origine u *et* dont l'extrémité v est dans `marked` ;

3. $r(v)$, lorsque $u \to v$ parcourt les arcs arrières ou transverses d'origine u dont l'extrémité v est dans la même composante fortement connexe que u. (On notera que v est alors nécessairement dans `marked`, car $r(v) < r(u)$, voir la table 4.)

On demande de plus qu'en entrée la formule $Pre(\texttt{stk}, \texttt{u})$ suivante soit satisfaite :

— si P est la pile dénotée par `stk`, alors $P :: \texttt{u} = \pi_k$, où $k \stackrel{\text{def}}{=} r(\texttt{u})$.

En sortie, on demande que la formule $Post(\texttt{stk}, \texttt{u})$ suivante soit satisfaite :

— si $\texttt{u} = a(\texttt{u})$, alors $Pre(\texttt{stk}, \texttt{u})$;

— sinon, alors `stk` dénote la pile π_k, où k est le rang maximal d'un sommet v dans la même composante fortement connexe que `u` et tel que $\texttt{u} \to_T^* v$.

Voici un aperçu sommaire de l'argument de correction.

En ligne 7, l'arc $\texttt{u} \to \texttt{v}$ est nécessairement un arc de liaison, comme dans `dfs_1`. En ligne 9, le test $\texttt{v} \in \texttt{stk}$ réussit si et seulement si `v` est dans π_k, où (grâce à la post-condition) k est dans un intervalle permettant d'utiliser la proposition 6.13, et donc, les sommets `v` qui réussissent ce test sont exactement ceux qui sont dans la même composante fortement connexe que `u`. À la fin de la boucle des lignes 5–9, on a donc établi l'invariant **(vi)** pour le sommet $u \stackrel{\text{def}}{=} \texttt{u}$.

En ligne 10, l'invariant **(vi)** se simplifie, car tous les sommets v tels que $\texttt{u} \to_T v$ sont dans `marked`. Ceci est dû au fait que $\texttt{dfsNum} = fin(v)$ à la fin de l'appel récursif $\texttt{scc}(v)$ (ligne 7), ceci étant dû à la post-condition de `scc`, et au fait que I_3 assure que `marked` contient tous les sommets de L_k avec $k \stackrel{\text{def}}{=} \texttt{dfsNum}$. On reconnaît alors en l'invariant **(vi)** les trois cas de la proposition 6.12. En conséquence, en ligne 10, $\texttt{low}(\texttt{u})$ est égal à $r(a(\texttt{u}))$. Comme dans l'algorithme `dfs_1`, $\texttt{r}(\texttt{u}) = r(\texttt{u})$. Donc le test `r(u)=low(u)` de la ligne 10 réussit si et seulement si `u` est le point d'entrée $*_C$ de sa composante fortement connexe C.

La propriété $Post(\texttt{stk}, \texttt{v})$ assurée par l'appel récursif de la ligne 7 (pour chaque `v` tel que $\texttt{u} \to \texttt{v}$ par un arc avant, donc) permet de conclure qu'à la ligne 10, `stk` représente la pile π_k où k est le rang maximal d'un sommet v dans la même composante fortement connexe que `u` et tel que $\texttt{u} \to_T^* v$. On trouve donc en suffixe de π_k tous les descendants de `u` dans T, par le lemme 4.8 (BBC). Si le test de la ligne 10 réussit, la pile π_k représentée par `stk` contient donc comme suffixe tous les sommets de C, avec $*_C$ comme premier sommet. Ceci assure la validité de $Post(\texttt{stk}, \texttt{u})$ en fin de procédure, que le test de la ligne 10 réussisse ou non.

Finalement, la complexité de `scc` est la même que `dfs_1`, à savoir $O(m + n)$, où n est le nombre de sommets de G et m son nombre d'arcs, à laquelle on doit rajouter la complexité induite par les lignes 10–11. Comme les opérations de pile sont en temps constant, cette complexité additionnelle est proportionnelle au nombre de sommets jamais dépilés de `stk` au cours de toute la procédure, et ceci vaut n, chaque composante fortement connexe C étant dépilée exactement une fois, et chaque sommet `v` de C étant dépilé exactement une fois.

```
  fun roy-warshall (M, n) =
1     A := copy(M);
2     for i=1..n
3         A [i, i] := 1;
4     for k=1..n
5         for i=1..n
6             for j=1..n
7                 A [i, j] := A [i, j] or
8                             (A [i, k] and A [k, j]);
9     return A;
```

TABLE 8 – L'algorithme de Roy-Warshall

Avec tout ceci, on conclut. On rappelle que l'hypothèse d'existence d'une source $*$ n'est pas restrictive.

Théorème 6.14 *L'algorithme de Tarkan* `scc` *calcule correctement les composantes fortement connexes d'un graphe G avec une source $*$: à la fin de l'algorithme, pour tout sommet u de G, $\mathtt{c}(u)$ vaut le point d'entrée $*_C$ de la composante fortement connexe C de u, et donc deux sommets u et v sont dans la même composante fortement connexe si et seulement si $\mathtt{c}(u) = \mathtt{c}(v)$. Sa complexité est en $O(m + n)$.*

7 Graphes valués, automates, distances minimales et maximales

Nous explorons de nouveaux algorithmes d'accessibilité. L'algorithme de Roy-Warshall calcule tous les couples de sommets (s, t) tels que $s \to^* t$, et fonctionne naturellement avec une représentation des graphes par matrice d'adjacence. Il se généralise à des questions similaires sur des graphes valués, permettant de résoudre des questions de théorie des langages, des calculs de probabilités, des calculs de distances maximales et minimales.

7.1 L'algorithme de Roy-Warshall

Une petite remarque avant de commencer : on prononce Roy comme « roi » ; Bernard Roy est français.

L'algorithme de Roy-Warshall, découvert indépendamment par Roy (B. Roy, *Transitivité et connexité*, C. R. Acad. Sci. Paris 249 :216–218, 1959) et Warshall (S. Warshall, *A Theorem on Boolean Matrices*, J. ACM 9(1) :11–12, 1962) est présenté en table 8. En entrée, il prend une matrice d'adjacence `M` d'un graphe G

à n sommets, et l'entier n. Les sommets de G sont donc implicitement les numéros $1, \ldots, n$. En sortie, il retourne la matrice d'adjacence `A` de la clôture réflexive transitive de G, autrement dit, pour tous $i, j \in \{1, \cdots, n\}$, $\mathtt{A}[i,]$ vaut 1 si $i \to^* j$, et 0 sinon.

L'idée approximative de cet algorithme est qu'au tour `k` de la boucle des lignes 4–8, la sous-matrice $(\mathtt{A}[i,j])_{1 \leq i,j \leq \mathtt{k}}$ est la clôture réflexive-transitive du sous-graphe de matrice d'adjacence $(\mathtt{M}[i,j])_{1 \leq i,j \leq \mathtt{k}}$. Ce n'est pas tout à fait exact, et nous verrons l'invariant que satisfait cet algorithme plus bas. Sa définition précise est assez subtile.

En attendant, il est clair que la complexité de cet algorithme est en $O(n^3)$. On notera que la recopie de la matrice à la ligne 1 n'est pas en temps constant, mais en temps $O(n^2)$, et que la boucle des lignes 2 et 3 est en temps $O(n)$, mais tout ceci est négligeable devant un $O(n^3)$.

Définition 7.1 *Pour tout graphe $G \stackrel{def}{=} (S, A)$ avec $S = \{1, \cdots, n\}$, pour tout $k \in \{1, \cdots, n\}$, on note G_k le graphe (S, A_k), où A_k est l'ensemble des couples $(i,j) \in S^2$ tels qu'il existe un chemin*

$$i = x_0 \to x_1 \to \cdots \to x_\ell = j$$

*avec $\ell \in \mathbb{N}$, dont l'*intérieur *$\{x_i \mid 0 < i < \ell\}$ est inclus dans $\{1, \cdots, k\}$.* intérieur

L'intérieur du chemin est égal à $\{x_1, \cdots, x_{\ell-1}\}$ si $\ell \geq 2$, et est vide si $\ell = 0$ ou $\ell = 1$. On ne requiert rien de l'origine i et de l'extrémité j des chemins dans A_k. Il est à noter que G_k n'est *pas* un graphe sur les k premiers sommets : l'ensemble des sommets de G_k est le même que celui de S.

Proposition 7.2 *Avec les notations de la définition 7.1, pour tous $(i,j) \in S^2$, on a :*

1. *$(i,j) \in A_0$ si et seulement si $i = j$ ou $i \to j$ dans G ;*
2. *pour tout $k \in \{1, \cdots, n\}$, $(i,j) \in A_k$ si et seulement si $(i,j) \in A_{k-1}$, ou bien (i,k) et (k,j) sont dans A_{k-1}.*

Démonstration. Le point 1 est évident : A_0 est l'ensemble des couples de sommets (i,j) tel qu'il y a un chemin de i à j dans G d'intérieur vide. Pour le point 2, la direction si est claire. Réciproquement, soit $(i,j) \in A_k$. Il existe un chemin $i = x_0 \to x_1 \to \cdots \to x_\ell = j$ dont l'intérieur est dans $\{1, \cdots, k\}$, et on peut le choisir de longueur ℓ minimale. Comme au lemme 4.3, ce chemin est alors élémentaire, donc il existe au plus un sommet intérieur égal à k. S'il n'y en a pas, alors $(i,j) \in A_{k-1}$. S'il y en a un, disons x_q, avec $0 < q < \ell$, alors $i = x_0 \to x_1 \to \cdots \to x_q = k$ est un chemin dont l'intérieur est inclus dans $\{1, \cdots, k-1\}$, donc $(i,k) \in A_{k-1}$, et $k = x_q \to x_{q+1} \to \cdots \to x_\ell = j$ est un chemin dont l'intérieur est inclus dans $\{1, \cdots, k-1\}$, donc $(k,j) \in A_{k-1}$. □

Les lignes 1–3 calculent donc une matrice `A` égale à A_0. On pourrait penser que les lignes 5–8 calculent A_k en fonction de A_{k-1}, mais elles le font *en place*. Le problème est le suivant. Disons que `A[i,j]` est *correct au rang* k si et seulement il vaut 1 si et seulement si $(i,j) \in A_k$, et 0 sinon. Aux lignes 7–8, le point 2 de la proposition 7.2 nous garantit que le calcul produit un `A[i,j]` correct au rang k à condition que l'ancienne valeur de `A[i,j]` soit correcte au rang $k-1$, et de même pour les valeurs de `A[i,k]` et de `A[k,j]` (pour `k` $= k$). Mais ces dernières ont pu être modifiées par de précédentes itérations des boucles imbriquées des lignes 5–8, et plutôt que d'être correctes au rang $k-1$, elles sont peut-être correctes au rang k, plutôt.

Le lemme suivant montre que la notion de correction au rang $k-1$ et au rang k est la même, ce qui élimine la difficulté — mais attention, ce sera la même uniquement pour les entrées `A[i,k]` et de `A[k,j]`, pas nécessairement les entrées à d'autres positions de la matrice `A`.

Lemme 7.3 *En reprenant les notations de la définition 7.1, pour tous* $k, i, j \in \{1, \cdots, n\}$*, on a :*

1. $(i,k) \in A_{k-1}$ *si et seulement si* $(i,k) \in A_k$ *;*
2. $(k,j) \in A_{k-1}$ *si et seulement si* $(k,j) \in A_k$*.*

Démonstration. Clairement, $A_{k-1} \subseteq A_k$. Si (i,k) est dans A_k, alors il existe un chemin de longueur minimale, donc élémentaire, de i à k et dont l'intérieur est dans $\{1, \cdots, k\}$. Comme il est élémentaire, son intérieur est en fait dans $\{1, \cdots, k-1\}$. On raisonne de même si $(k,j) \in A_k$. □

On peut maintenant donner une preuve de correction en démontrant les invariants suivants à la ligne 7.

1. pour tout $(i,j) \in S^2$ lexicographiquement plus petit ou égal à $(\mathtt{i}, \mathtt{j})$, $\mathtt{A}[i,j] =$ 1 si $(i,j) \in A_{\mathtt{k}}$, et 0 sinon ;
2. pour tout $(i,j) \in S^2$ non lexicographiquement plus petit ou égal à $(\mathtt{i}, \mathtt{j})$, $\mathtt{A}[i,j] = 1$ si $(i,j) \in A_{\mathtt{k}-1}$, et 0 sinon.

Ceci est laissé en exercice au lecteur. À la fin de l'algorithme, on trouve dans le tableau `A`, à chaque entrée (i,j), la valeur 1 si et seulement si $(i,j) \in A_{\mathtt{n}}$, si et seulement s'il existe une chemin de i à j dont l'intérieur est inclus dans $\{1, \cdots, \mathtt{n}\}$. Mais cette condition sur l'intérieur est toujours vraie, car tout sommet du graphe donné en entrée est de numéro entre 1 et `n`. On en déduit :

Théorème 7.4 *L'algorithme de Roy-Warshall* `roy-warshall` *calcule la clôture réflexive-transitive d'un graphe* $G \stackrel{def}{=} (\{1, \cdots, n\}, A)$*, lorsqu'on lui fournit en entrée sa matrice d'adjacence et* n*. Sa complexité est en* $O(n^3)$*.*

7.2 Graphes valués

Un graphe valué est un graphe avec une information supplémentaire sur chaque arc, sous forme d'une valeur dans un domaine adéquat. Un *monoïde* est un triplet $(K, \times, 1)$ où K est un ensemble, $\times$ est une loi interne associative sur K, et 1 est un élément neutre pour $\times$. Il est *commutatif* si et seulement si $\times$ est commutative. On préfère souvent la notation $(K, +, 0)$ pour un monoïde commutatif. Lorsque la loi interne et le neutre sont claires, on notera juste K pour un monoïde.

monoïde

commutatif

Définition 7.5 *Soit K un monoïde. Un* graphe valué*, à valeurs dans K, est un couple (G, V) où $G \stackrel{def}{=} (S, A)$ est un graphe et $V \colon A \to K$ est une fonction de l'ensemble A des arcs vers un ensemble K. Les éléments de K sont appelés les* valeurs*.*

graphe valué

valeurs

Un graphe valué dont les sommets sont numérotés (donc $S = \{1, \cdots, n\}$) se représente sous forme d'une matrice, que l'on continuera d'appeler matrice d'adjacence. Informellement, au lieu d'y écrire des 0 et des 1, chaque 1 à la position (i, j) sera remplacé par $V(i, j)$. Nous adapterons cette définition plus bas.

On notera $u \xrightarrow{a} v$ un arc (u, v) de poids $a \stackrel{\text{def}}{=} V(u, v)$.

Définition 7.6 *Le* poids *d'un chemin $x_0 \xrightarrow{a_1} x_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} x_k$ est le produit $a_1 \times a_2 \times \cdots \times a_k$.*

poids

Si $k = 0$, ce poids vaut par convention l'élément neutre 1 du monoïde K. Voici quelques exemples.

1. Si K est le monoïde $(\mathbb{N}, +, 0)$ ou $(\mathbb{R}, +, 0)$, un poids s'interprète comme une distance ;
2. si $K = ([0, 1], \times, 1)$, un poids s'interprète comme une probabilité ;
3. si $K = (\mathbb{P}(\Sigma^*), \cdot, \{\epsilon\})$, où Σ est un alphabet, ϵ est le mot vide, et $A \cdot B \stackrel{\text{def}}{=} \{uv \mid u \in A, v \in B\}$ pour tous langages $A, B \in \mathbb{P}(\Sigma^*)$, un poids est un langage, et un graphe valué peut être vu comme un automate fini (généralisé : le poids d'un arc dans un automate fini ordinaire est juste un singleton $\{a\}$, $a \in \Sigma$).

Pour combiner les poids de plusieurs chemins, on aura besoin que K ait davantage de structure.

Définition 7.7 *Un* semi-anneau *est un quintuplet $(K, +, 0, \times, 1)$ où :*

semi-anneau

— *$(K, +, 0)$ est un monoïde commutatif ;*

— *$(K, \times, 1)$ est un monoïde ;*

— *les lois de distributivité suivantes sont valides :*

$$a \times (b + c) = (a \times b) + (a \times c) a \times 0 = 0$$
$$(b + c) \times a = (b \times a) + (c \times a) 0 \times a = 0,$$

pour tous $a, b, c \in K$.

On notera souvent juste K pour un semi-anneau, comme avec les monoïdes. Pour tout ensemble *fini* C de chemins, on peut alors définir la somme des poids des chemins de C. Voici quelques exemples :

1. $(\mathbb{R}_+, +, 0, \times 1)$ est un semi-anneau de « probabilités » ; les guillemets sont là pour attirer l'attention que chaque chemin est considéré (peut-être à tort) comme un événement disjoint.
2. $(\mathbb{N} \cup \{-\infty\}, \max, -\infty, +, 0)$, le *semi-anneau arctique* : la somme est un max, le max de l'ensemble vide étant pris égal à $-\infty$ par convention, et la somme des poids des chemins de C s'interprète comme une distance maximale ; ceci a un sens même si C est infini. semi-anneau arctique
3. $(\mathbb{N} \cup \{+\infty\}, \min, +\infty, +, 0)$ est le *semi-anneau tropical*, et sert à calculer des distances minimales. semi-anneau tropical
4. $(\mathbb{P}(\Sigma^*), \cup, \emptyset, \cdot, \{\epsilon\})$: ici aussi, la somme des poids des chemins de C a un sens même si C est infini, et est l'union des langages obtenus comme poids des chemins de C. Ceci généralise la notion de langage défini par un automate fini.

Nous en arrivons à la définition d'une matrice d'adjacence pour un graphe valué, à valeurs dans un semi-anneau.

Définition 7.8 *La* matrice d'adjacence *d'un graphe valué* (S, A, V)*, avec* $S \stackrel{def}{=} \{1, \cdots, n\}$*, où* V *est une fonction de* A *vers un semi-anneau* $(K, +, 0, \times, 1)$*, est la matrice* $n \times n$ *dont chaque entrée* (i, j) *vaut :* matrice d'adjacence

— $V(i, j)$ *si* $(i, j) \in A$ *;*

— 0 *sinon.*

Il s'agit d'une représentation ambiguë : il n'y a aucun moyen de savoir, à partir de la matrice d'adjacence, s'il y a un arc $u \xrightarrow{0} v$ ou pas d'arc de u à v. Notre but étant de calculer des sommes de poids de chemins dans un ensemble C, ceci n'aura aucune importance.

7.3 Algèbres de Kleene continues

Lorsque C est fini, la somme des poids des chemins de C est bien définie, si K est un semi-anneau. Dans le cas général, nous aurons besoin donner un sens à certaines sommes infinies.

Définition 7.9 *Un monoïde $(K, +, 0)$ est* idempotent *si et seulement si $a+a = a$ pour tout $a \in K$. Dans ce cas, on pose $a \leq b$ si et seulement si $b = a + b$, pour tous $a, b \in K$.* idempotent

Un semi-anneau $(K, +, 0, \times, 1)$ est idempotent *si et seulement si $(K, +, 0)$ est un monoïde idempotent.* idempotent

Lemme 7.10 *Pour tout monoïde commutatif idempotent $(K, +, 0)$, $(K, \leq)$ est un* semi-treillis supérieur *: $\leq$ est une relation d'ordre, $a+b$ est la borne supérieure de $\{a, b\}$ pour tous $a, b \in K$, et 0 est le plus petit élément de K.* semi-treillis supérieur

Démonstration. On a $a = a + a$ par idempotence, donc $a \leq a$. Si $a \leq b$ et $b \leq c$, alors par définition $b = a+b$ et $c = b+c$, donc $c = (a+b)+c = a+(b+c) = a+c$, ce qui démontre que $a \leq c$. Si $a \leq b$ et $b \leq a$, alors par définition $b = a + b$ et $a = b + a$. Par commutativité, $b = a$. Donc $\leq$ est une relation d'ordre.

On a $a \leq a + b$, puisque $a + b = a + (a + b)$, par associativité et idempotence. De même, et en utilisant la commutativité, $b \leq a + b$. Si c est un majorant de $\{a, b\}$, on a $c = a+c$ et $c = b+c$, par définition, donc $c = a+(b+c) = (a+b)+c$, ce qui montre que $a + b \leq c$. Donc $a + b$ est le plus petit des majorants de $\{a, b\}$.

Finalement, on a $0 \leq a$ pour tout $a \in K$, car $a = 0 + a$. □

Réciproquement, on peut montrer que tout semi-treillis supérieur $(K, \leq)$ est un monoïde commutatif idempotent, muni de l'opération qui à tout couple d'éléments (a, b) associe leur borne supérieure.

On notera ab plutôt que $a \times b$ dans la suite. La notation b^n représente le produit de n copies de b ; si $n = 0$, $b^n = 1$.

Définition 7.11 *Une* algèbre de Kleene continue *est un semi-anneau idempotent $(K, +, 0, \times, 1)$ tel que :* algèbre de Kleene continue

1. *toute famille de la forme $(b^n)_{n\in\mathbb{N}}$ a une borne supérieure, que l'on notera b^* ;*
2. *pour toute famille dénombrable $(u_n)_{n\in\mathbb{N}}$ ayant une borne supérieure $\sup_{n\in\mathbb{N}} u_n$, pour tous $a, b \in K$, la famille $(au_nb)_{n\in\mathbb{N}}$ a aussi une borne supérieure, et $\sup_{n\in\mathbb{N}} au_nb = a(\sup_{n\in\mathbb{N}} u_n)b$.*

La notion de borne supérieure a un sens car K est ordonné par $\leq$, grâce au lemme 7.10. La condition 2 est la condition de *continuité*.

Remarque 7.12 *La condition de continuité exprime que le produit est continu, au sens où il commute aux bornes supérieures existantes. L'addition est toujours continue, en ce même sens, car l'addition est une borne supérieure (lemme 7.10), et les bornes supérieures commutent toujours.*

Remarque 7.13 *Dans la littérature, une algèbre de Kleene est un sextuplet $(K, +, 0, \times, 1, \_^*)$ où $(K, +, 0, \times, 1)$ est un semi-anneau, et $\_^*$ est une opération satisfaisant aux* axiomes de Kozen *:* axiomes de Kozen

— $1 + aa^* \leq a^*$;

— $1 + a^*a \leq a^*$;

— $ab \leq b$ *implique* $a^*b \leq b$;

— $ba \leq b$ *implique* $ba^* \leq b$,

pour tous $a, b \in K$*. Elle est appelée* $*$*-continue si et seulement si* $a^* = \sup_{n\in\mathbb{N}} a^n$ *pour tout* $n \in \mathbb{N}$ *; il existe des algèbres de Kleene non* $*$*-continues. Les algèbres de Kleene continues, telles que nous les avons définies, sont toutes* $*$*-continues, et ce sont les seules dont nous aurons besoin.*

Voici quelques exemples :

1. $(\mathbb{P}(\Sigma^*), \cup, \emptyset, \cdot, \{\epsilon\})$: l'opération _* est l'étoile de Kleene usuelle, autrement dit A^* est la collection des concaténations $w_1 \cdots w_n$ de mots $w_i \in A$, en nombre $n \in \mathbb{N}$ arbitraire.
2. $(Reg(\Sigma), \cup, \emptyset, \cdot, \{\epsilon\})$, où $Reg(\Sigma)$ est l'ensemble des langages rationnels (ou réguliers) sur Σ, avec l'étoile de Kleene.
3. $(\{0, 1\}, \max, 0, \min, 1)$: c'est l'*algèbre de Kleene triviale* ; on a $1^* = 1$, et $0^* = 1$. (0^* est le sup de $0^0 = 1$, $0^1 = 0$, $0^2 = 0$, etc. Ne pas oublier 0^0 !)

algèbre de Kleene triviale

Généralisons la notion de somme finie d'éléments d'un monoïde commutatif à des sommes potentiellement infinies dans une algèbre de Kleene continue. On profite du fait qu'une somme finie dans une algèbre de Kleene est une borne supérieure, par le lemme 7.10, ce qui nous permet de voir la définition suivante comme une généralisation de la notion de somme finie déjà introduite.

Définition 7.14 *Dans un semi-anneau, on appelle* somme *d'un ensemble* C *d'éléments (fini ou non) la borne supérieure de* C*, si elle existe.*

somme

7.4 L'algorithme de Roy-Warshall généralisé

L'algorithme de Roy-Warshall généralisé (aux graphes valués, à valeurs dans une algèbre de Kleene continue K) est donné en table 9. En entrée, il prend une matrice d'adjacence `M` pour un graphe valué G à valeurs dans une algèbre de Kleene continue K, et son nombre de sommets. Il retourne une matrice `A`, dont nous verrons plus bas qu'elle contient à toute entrée (i, j) la somme des points de tous les chemins d'origine i et d'extrémité j. On notera l'utilisation de l'opération `+`, de la multiplication, et de l'opération _* de l'algèbre de Kleene continue continue aux lignes 3, 5, 8 et 9.

Remarque 7.15 *Dans le cas de l'algèbre de Kleene triviale,* $(\{0, 1\}, \max, 0, \min, 1)$*, l'addition est un « ou », le produit est un « et », et* `A[k,k]*` *vaut toujours* 1*. On retrouve donc l'algorithme de Roy-Warshall usuel (table 8) dans ce cas.*

```
   fun roy-warshall-gen (M, n) =
1      A := copy(M);
2      for i=1..n
3          A [i, i] := A [i, i]+1;
4      for k=1..n
5          star := A[k,k]*;
6          for i=1..n
7              for j=1..n
8                  A [i, j] := A [i, j] +
9                              (A [i, k] × star × A [k, j]);
10     return A;
```

TABLE 9 – L'algorithme de Roy-Warshall généralisé aux graphes valués

Pour démontrer la correction de `roy-warshall-gen`, on raisonne comme pour `roy-warshall`.

Définition 7.16 *Pour tout graphe valué $G \stackrel{def}{=} (S, A, V)$, avec valeurs dans une algèbre de Kleene continue K et $S = \{1, \cdots, n\}$, pour tout $k \in \{1, \cdots, n\}$, on note V_k la fonction qui à tout couple $(i, j) \in S^2$ associe la somme (si elle existe) des poids des chemins de la forme :*

$$i = x_0 \to x_1 \to \cdots \to x_\ell = j$$

*avec $\ell \in \mathbb{N}$, et dont l'*intérieur $\{x_i \mid 0 < i < \ell\}$ *est inclus dans $\{1, \cdots, k\}$.* intérieur

Nous allons voir tout de suite que cette somme $V_k(i, j)$ existe toujours. L'analogue de la proposition 7.2 est la proposition suivante.

Proposition 7.17 *Avec les notations de la définition 7.16, pour tous $(i, j) \in S^2$, on a :*

1. *$V_0(i, j)$ est définie, et vaut $V(i, i) + 1$ si $i = j$, $V(i, j)$ sinon ;*
2. *pour tout $k \in \{1, \cdots, n\}$, $V_k(i, j)$ est bien définie et vaut $V_{k-1}(i, j) + (V_{k-1}(i, k) \times V_{k-1}(k, k)^* \times V_{k-1}(k, j))$.*

Démonstration. Pour le point 1, les chemins de i à j d'intérieur vide sont les arcs $i \to j$, plus le chemin de longueur nulle de i à i si $i = j$; et ce dernier est de poids 1.

Le point 2 est prouvé par récurrence sur k. (Ou plutôt, on montre par récurrence sur $k \in \{0, \cdots, n\}$ que $V_k(i, j)$ est bien défini pour tout $(i, j) \in S^2$, et que si $k \geq 1$ alors la formule donnée au point 2 pour $V_k(i, j)$ est correcte. Le cas $k = 0$ est donné par le point 1, et nous nous concentrons sur le cas $k \geq 1$, en supposant par hypothèse de récurrence que $V_{k-1}(i, j)$ est bien définie pour tout

$(i,j) \in S^2$.) Les chemins de i à j d'intérieur inclus dans $\{1, \cdots, k\}$ s'écrivent de façon unique :

$$i = x_0 \xrightarrow{a_1}{}^* x_1 \xrightarrow{a_2}{}^* \cdots \xrightarrow{a_{\ell-1}}{}^* x_{\ell-1} \xrightarrow{a_\ell}{}^* x_\ell = j, \tag{2}$$

où x_1, …, $x_{\ell-1}$ énumèrent les sommets égaux à k dans l'intérieur du chemin. On note par ailleurs en abrégé $u \xrightarrow{a}{}^* v$ un chemin de poids a de u à v. Parmi les chemins (2), on trouve ceux tels que $\ell = 0$ (aucun sommet égal à k dans l'intérieur), et leur somme vaut $V_{k-1}(i,j)$ par définition. Les chemins (2) obtenus avec un $\ell \geq 1$ sont les concaténations :

— de chemins de $i = x_0$ à $x_1 = k$ d'intérieur inclus dans $\{1, \cdots, k-1\}$, dont la somme des poids vaut $V_{k-1}(i,k)$;

— de $\ell - 1$ chemins de k à k d'intérieur inclus dans $\{1, \cdots, k-1\}$;

— de chemins de $x_{\ell-1} = k$ à $x_\ell = j$ d'intérieur inclus dans $\{1, \cdots, k-1\}$, dont la somme des poids vaut $V_{k-1}(k,j)$.

En distribuant les sommes (=les bornes supérieures), et grâce à la condition de continuité de l'algèbre de Kleene continue, la somme des chemins (2), lorsque $\ell \geq 1$ est fixé, existe et vaut donc $V_{k-1}(i,k)V_{k-1}(k,k)^{\ell-1}V_{k-1}(k,j)$. La somme $V_k(i,j)$ de tous les chemins (2), pour tout $\ell \in \mathbb{N}$, existe et vaut donc :

$$\begin{aligned} &V_{k-1}(i,j) + \sup_{\ell \geq 1} V_{k-1}(i,k)V_{k-1}(k,k)^{\ell-1}V_{k-1}(k,j) \\ &= V_{k-1}(i,j) + (V_{k-1}(i,k) \times V_{k-1}(k,k)^* \times V_{k-1}(k,j)) \end{aligned}$$

par définition de l'étoile, par la remarque 7.12 et par la condition de continuité. □

Comme pour `roy-warshall`, les modifications de `A` sont effectuées en place. On a donc une difficulté similaire à résoudre. L'analogue du lemme 7.3, qui justifie que les modifications en place effectuées aux lignes 8–9 ne causent aucun problème, cependant, n'est pas que $V_{k-1}(i,k) = V_k(i,k)$ ou que $V_{k-1}(k,j) = V_k(k,j)$. À la place, on raffine le point 2 de la proposition 7.17, voir le lemme 7.19 ci-dessous. On aura besoin de quelques remarques d'abord.

Lemme 7.18 *Dans un semi-anneau K,*

1. *si $A \subseteq B \subseteq K$ et si les sommes (les bornes supérieures) de A et de B existent, alors la somme de A est inférieure ou égale à celle de B ;*
2. *l'addition est monotone : si $a \leq b$ alors $a + c \leq b + c$;*
3. *le produit est monotone : si $a \leq b$ alors $ac \leq bc$ et $ca \leq cb$.*

Démonstration. 1. C'est un fait général que dans tout ensemble ordonné, si $A \subseteq B$ et $\sup A$, $\sup B$ existent, alors $\sup A \leq \sup B$. En effet, $\sup B$ est un majorant de B, et donc un majorant de son sous-ensemble A ; il est donc plus grand ou égal que le plus petit, $\sup A$, des majorants de A.

2. Comme $a \leq b$, on a $b = a + b$, donc $b + c = (a + b) + c = (a + b) + (c + c) = (a + c) + (b + c)$, donc $a + c \leq b + c$.

3. Comme $a \leq b$, on a $b = a + b$, donc $bc = (a + b)c = ac + bc$, donc $ac \leq bc$, et de même $ca \leq cb$. □

Lemme 7.19 *En reprenant les notations de la définition 7.16, pour tous $k, i, j \in \{1, \cdots, n\}$, on a :*

$$V_{k-1}(i,j) \leq V_k(i,j)$$
$$V_k(i,j) = V_{k-1}(i,j) + (a \times V_{k-1}(k,k)^* \times b)$$

pour tous $a, b \in K$ tels que $V_{k-1}(i,k) \leq a \leq V_k(i,k)$ et $V_{k-1}(k,j) \leq b \leq V_k(k,j)$.

Démonstration. Tous les chemins de i à j dont l'intérieur est inclus dans $\{1, \cdots, k-1\}$ ont un intérieur inclus dans $\{1, \cdots, k\}$. Donc $V_{k-1}(i,j) \leq V_k(i,j)$ en utilisant le point 1 du lemme 7.18.

Supposons maintenant $V_{k-1}(i,k) \leq a \leq V_k(i,k)$ et $V_{k-1}(k,j) \leq b \leq V_k(k,j)$. On a donc :

$$\begin{aligned}
V_k(i,j) &= V_{k-1}(i,j) + (V_{k-1}(i,k) \times V_{k-1}(k,k)^* \times V_{k-1}(k,j)) && \text{(proposition 7.17, point 2)} \\
&\leq V_{k-1}(i,j) + (a \times V_{k-1}(k,k)^* \times b) && \text{(monotonie, lemme 7.18)} \\
&\leq V_{k-1}(i,j) + (V_k(i,k) \times V_{k-1}(k,k)^* \times V_k(k,j)) && \text{(monotonie, lemme 7.18)} \\
&\leq V_k(i,j),
\end{aligned}$$

car l'avant-dernière somme est une somme de poids de chemins de i à j dont les intérieurs sont tous dans $\{1, \cdots, k\}$. Tous les termes de la suite d'inégalités ci-dessus sont donc égaux, en particulier ceux des deux premières lignes. □

Les invariants à établir en début de ligne 5 sont similaires à ceux utilisés pour démontrer la correction de `roy-warshall` :

1. pour tout $(i,j) \in S^2$ lexicographiquement plus petit ou égal à $(\mathtt{i}, \mathtt{j})$, $\mathtt{A}[i,j] = V_k(i,j)$;
2. pour tout $(i,j) \in S^2$ non lexicographiquement plus petit ou égal à $(\mathtt{i}, \mathtt{j})$, $\mathtt{A}[i,j] = V_{k-1}(i,j)$.

On utilise le lemme 7.19 pour montrer que c'est effectivement un invariant.

À la fin de l'algorithme, on a $\mathtt{A}[i,j] = V_{\mathtt{n}}(i,j)$ pour tout $(i,j) \in S^2$. Or $V_{\mathtt{n}}(i,j)$ est la somme des poids des chemins de i à j d'intérieur inclus dans $\{1, \cdots, \mathtt{n}\}$, et cette condition sur l'intérieur est simplement vraie. On en déduit le théorème suivant.

Théorème 7.20 *Pour tout graphe valué $G \stackrel{def}{=} (\{1, \cdots, n\}, A, V)$, où V est à valeurs dans une algèbre de Kleene continue K, l'algorithme de Roy-Warshall généralisé `roy-warshall-gen` appelé sur la matrice d'adjacence de G et n retourne un tableau `A` tel que, pour tous $i, j \in \{1, \cdots, n\}$ est la somme des poids de tous les chemins de i à j dans G. La complexité de l'algorithme est en $O(n^3)$, en supposant les opérations de l'algèbre de Kleene continue K en temps constant.*

L'hypothèse du temps constant pour les opérations de K est importante, comme on va le voir à la section suivante.

7.5 L'algorithme de McNaughton-Yamada

Un automate fini $\mathcal{A}$ peut être représente par la donnée de son état initial, de l'ensemble de ses états finaux, et d'un graphe valué $G \stackrel{\text{def}}{=} (V, A, S)$, où V est l'ensemble des états, il y a un arc $q \to q'$ par transition $q \xrightarrow{a} q'$ de l'automate, et V envoie un tel arc vers le langage $\{a\} \in \mathbb{P}(\Sigma^*)$. L'algèbre de Kleene continue est celle des langages rationnels. Informatiquement, on représente ces langages rationnels par des expressions régulières.

Dans ce contexte, l'algorithme de Roy-Warshall généralisé calcule un tableau V^*, où pour tout couple d'états q et q' (dans V), $V^*(q, q')$ est le langage des mots reconnus en q' si l'on considère q comme l'état initial.

Ce cas spécifique de l'algorithme de Roy-Warshall généralisé s'appelle l'*algorithme de McNaughton-Yamada* (R. McNaughton et Hisao Yamada, *Regular Expressions and State Graphs for Automata*, IRE Trans. Electronic Computers, EC-9(1) :39–47, 1960). Il permet de montrer le théorème bien connu suivant.

algorithme de McNaughton-Yamada

Théorème 7.21 *Il existe un algorithme qui convertit tout automate fini $\mathcal{A}$ en une expression régulière dont le language est identique au langage des mots acceptés par $\mathcal{A}$.*

La réciproque est vraie, mais sort du cadre de ce cours.

On fera attention, cependant, au fait que les opérations sur les expressions régulières ($+$, $\times$, $\_^*$) ne sont pas nécessairement en temps constant. Dans une réalisation naïve, par exemple, une expression régulière serait une chaîne de caractères, et calculer $a + b$ demanderait à recopier a, ajouter le symbole « $+$ », puis recopier b, produisant une expression plus grosse que a et b. Ce problème est inévitable si l'on souhaite fournir une expression régulière en forme de chaîne de caractères en sortie, même en utilisant des structures de données bien étudiées pour calculer les résultats intermédiaires : un résultat de H. Gruber et M. Holzer (*From Finite Automata to Regular Expressions and Back—A Summary on Descriptional Complexity*. Int. J. Found. Comp. Sci. 26(8) :1009-1040, 2015) montre que la complexité de l'algorithme de McNaughton-Yamada est en $\Theta(|\Sigma|.4^n)$, où Σ est l'alphabet est n est le nombre d'états de l'automate. Comme cet algorithme effectue $O(n^3)$ opérations élémentaires, ces opérations élémentaires ne peuvent pas être effectuées en temps polynomial en n.

En revanche, on peut représenter les expressions régulières par des arbres de syntaxe, ce qui est naturel en Caml, et chaque opération sur K est alors en temps constant. Dans ce cas, l'algorithme de McNaughton-Yamada retourne un arbre de syntaxe d'une expression régulière en temps $O(n^3)$. Mais cet arbre peut décrire une expression régulière de taille exponentielle en n, et ceci s'explique par le fait que de nombreux sous-arbres seront nécessairement partagés.

7.6 L'algorithme de Floyd

L'algorithme de Floyd (R. W. Floyd, *Algorithm 97 : Shortest Path*, Comm. ACM 5(6) :345, 1962) est le cas particulier de l'algorithme `roy-warshall-gen` où le semi-anneau considéré est $(\mathbb{R} \cup \{-\infty, +\infty\}, \max, -\infty, +, 0)$, et où les poids des arcs des graphes considérés ne valent jamais ni $-\infty$ ni $+\infty$. On prend comme convention :

Convention 7.22 (Semi-anneau des distances max) $(-\infty)+(+\infty) = -\infty$.

Cette convention est nécessaire d'une part pour que $(\mathbb{R}\cup\{-\infty, +\infty\}, \max, -\infty, +, 0)$ soit effectivement un semi-anneau ($-\infty$ doit être le zéro pour la multiplication $+$). Le fait que le poids des arcs ne vaut jamais $-\infty$ ne change rien à l'affaire : dans une matrice d'adjacence, les entrées (i, j) ne correspondant à aucun arc valent le zéro du semi-anneau, qui est justement $-\infty$; par ailleurs, l'algorithme ci-dessous fabriquera des matrices dont les entrées peuvent valoir $+\infty$.

L'ordre associé est l'ordre usuel $\leq$. Il s'agit d'une algèbre de Kleene continue, où $a^* = \sup(0, a, 2a, 3a, \cdots)$, donc $a^* = 0$ si $a \leq 0$, $a^* = +\infty$ sinon.

Le poids d'un chemin $x_0 \xrightarrow{a_1} x_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} x_k$ est égal à $a_1 + a_2 + \cdots a_k$, et s'interprète comme une distance. L'algorithme de Floyd (=Roy-Warshall généralisé) calcule alors, pour tout couple de sommets (i, j), la borne supérieure des distances (=poids) de i à j. On dit souvent que l'algorithme de Floyd calcule les *distances max* entre tout couple de sommets.

Proposition 7.23 (Poids strictement positifs) *Soit $G \stackrel{def}{=} (S, A, V)$ un graphe valué, à valeurs dans $(\mathbb{R} \cup \{-\infty, +\infty\}, \max, -\infty, +, 0)$, et supposons que V soit à valeurs dans $\mathbb{R}$.*

La borne supérieure des distances de tout sommet x à tout sommet y est atteinte, et est donc un max*, si et seulement si :* max

1. *il existe un chemin $x \to^* y$,*
2. *et pour tout sommet z tel que $x \to^* z \to^* y$ dans G, il n'y a pas de circuit $z \to^+ z$ de poids strictement positif.*

Si la condition 1 n'est pas vérifiée, le supremum vaut $-\infty$, et si la condition 1 est vérifiée mais pas la condition 2, le supremum vaut $+\infty$.

Démonstration. S'il n'existe pas de chemin $x \to^* y$, alors la borne supérieure de l'ensemble (vide) des distances de x à y vaut $-\infty$. Supposons donc la condition 1 satisfaite. S'il existe un chemin $x \to^* z \to^* y$ est un circuit $z \to^+ z$ de poids strictement positif, alors il y a des chemins $x \to^* z(\to^+ z)^n \to^* y$ pour n arbitrairement grand, et la borne supérieure de leurs poids vaut $+\infty$.

Supposons maintenant les conditions 1 et 2 satisfaites. Pour tout chemin $\gamma\colon x \to^* y$, soit γ' le chemin obtenu en éliminant les circuits de γ. Comme tous les circuits sont de poids négatif ou nul, le poids de γ' est inférieur ou égal à celui de γ. La famille des poids de chemins *élémentaires* de x à y est donc cofinale dans

```
   fun floyd-max (M, n) =
1      A := copy(M);
2      for i=1..n
3          A [i, i] := max(A [i, i], 0);
4      for k=1..n
5          if A[k,k]<=0 then star:=0 else star:=+∞;
6          for i=1..n
7              for j=1..n
8                  A [i, j] := max (A [i, j],
9                                   (A [i, k] + star + A [k, j]);
10     return A;
```

TABLE 10 – L'algorithme de Floyd de calcul des distances maximales

celle des poids de tous les chemins de x à y, et a donc la même borne supérieure. Finalement, il n'y a qu'un nombre fini de chemins élémentaires de x à y, et comme l'ordre du $\mathbb{R} \cup \{-\infty, +\infty\}$ est total, cette borne supérieure est atteinte en un de ces chemins élémentaires. □

Explicitement, l'algorithme de Floyd de calcul des distances maximales (ou $+\infty$) entre tout couple de sommets d'un graphe valué, à valeurs dans $\mathbb{R}$, donné par sa matrice d'adjacence, est donné en table 10. Sa correction, et le fait que sa complexité soit en $O(n^3)$, est un cas particulier du théorème 7.20. On doit bien faire attention à implémenter l'opération + de sorte à respecter la convention 7.22.

On peut optimiser cet algorithme. Par exemple, si l'on découvre à la ligne 5 que $\texttt{A}[\texttt{k},\texttt{k}] > \texttt{0}$, alors il sera un peu plus efficace de brutalement affecter $+\infty$ à toutes les entrées `A [i, j]` différentes de $-\infty$. Dans le cas où $\texttt{A}[\texttt{k},\texttt{k}] <= \texttt{0}$, la somme avec `star` à la ligne 9 n'est pas nécessaire. Ceci mène à l'algorithme de la table 11. Sa complexité théorique est toujours en $O(n^3)$.

En général, l'algorithme de Floyd pour les distances maximales est donné sous la forme simplifiée de la table 12, qui ne fonctionne que si le graphe valué en entrée n'a aucun circuit de poids strictement positif.

De façon entièrement symétrique, on peut calculer des *distances min*. Le semi-anneau considéré est alors $(\mathbb{R} \cup \{-\infty, +\infty\}, \min, +\infty, +, 0)$, avec une la convention différente suivante.

Convention 7.24 (Semi-anneau des distances min) $(-\infty)+(+\infty) = +\infty$.

L'ordre associé est désormais l'*opposé* $\geq$ de l'ordre $\leq$. Il s'agit d'une algèbre de Kleene continue, où $a^* = 0$ si $a \geq 0$, et $-\infty$ sinon. L'algorithme de Floyd est l'algorithme de Floyd-Warshall généralisé, spécialisé à ce semi-anneau, et calcule les *distances min* entre tout couple de sommets. La proposition suivante se démontre comme la proposition 7.23.

```
fun floyd-max-opt (M, n) =
    A := copy(M);
    for i=1..n
        A [i, i] := max(A [i, i], 0);
    for k=1..n
        if A[k,k]>0
           then for i=1..n
                    for j=1..n
                        if A [i, j]!=−∞ then A [i, j]:=+∞
        else for i=1..n
                 for j=1..n
                     A [i, j] := max (A [i, j],
                                 (A [i, k] + A [k, j]);
    return A;
```

TABLE 11 – L'algorithme de Floyd de calcul des distances maximales, optimisé

```
fun floyd-max-simple (M, n) =
    A := copy(M);
    for i=1..n
        A [i, i] := max(A [i, i], 0);
    for k=1..n
        for i=1..n
            for j=1..n
                A [i, j] := max (A [i, j],
                            (A [i, k] + A [k, j]);
    return A;
```

TABLE 12 – L'algorithme de Floyd de calcul des distances maximales, pour le cas des graphes valués sans circuit de poids strictement positif

Proposition 7.25 (Poids strictement négatifs) *Soit* $G \stackrel{def}{=} (S, A, V)$ *un graphe valué, à valeurs dans* $(\mathbb{R} \cup \{-\infty, +\infty\}, \min, +\infty, +, 0)$*, et supposons que* V *soit à valeurs dans* $\mathbb{R}$*.*

La borne inférieure des distances de tout sommet x *à tout sommet* y *est atteinte, et est donc un* min*, si et seulement si :* min

1. *il existe un chemin* $x \to^* y$*,*
2. *et pour tout sommet* z *tel que* $x \to^* z \to^* y$ *dans* G*, il n'y a pas de circuit* $z \to^+ z$ *de poids strictement négatif.*

Si la condition 1 n'est pas vérifiée, l'infimum vaut $+\infty$*, et si la condition 1 est vérifiée mais pas la condition 2, l'infimum vaut* $-\infty$*.*

On laisse au lecteur le soin de réécrire des variantes des algorithmes `floyd-max` (table 10), `floyd-max-opt` (table 11) et `floyd-max-simple` (table 12) pour calculer les distances min plutôt que max. Il suffit d'échanger $+\infty$ et $-\infty$, de remplacer `max` par `min`, et de ne pas oublier que + doit être implémenté avec la convention 7.24, et non plus la convention 7.22.

Nous verrons un meilleur algorithme que l'algorithme de Floyd, l'algorithme de *Johnson*, à la section 8.8. Il s'agit d'une combinaison astucieuse des algorithmes de Bellman-Ford et de Dijkstra, que nous verrons en section 8.

Remarque 7.26 *Nous avons vu avec l'algorithme de McNaughton-Yamada qu'il fallait faire attention à la complexité des opérations du semi-anneau utilisé. L'algorithme de Floyd est souvent considéré comme s'exécutant en temps* $O(n^3)$*, et ceci suppose que les opérations arithmétiques faites sur les valeurs (réelles) des arcs sont en temps constant. En pratique, ces poids sont souvent des entiers machine, et cette hypothèse est alors vérifiée. Mais l'opération d'addition* $\mathtt{A[i,k]} + \mathtt{A[k,j]}$ *peut alors causer des débordements arithmétiques, qui doivent être gérés, voire anticipés. (Par « anticipé », on veut dire qu'on peut vérifier qu'il n'y aura pas de débordement arithmétique si la plus grand valeur d'un arc ne dépasse pas* M/n*, où* M *est le plus grand entier représentable, par exemple.) Si l'on souhaite éviter les débordements arithmétiques, on peut utiliser des grands entiers, comme on en trouve nativement en Python ou via des bibliothèques comme la* `gmp` *dans d'autres langages, mais les opérations arithmétiques ne sont plus en temps constant. La situation est encore pire si les valeurs sont rationnelles, ou à l'extrême si l'on utilise des* réels exacts *(voir par exemple H.-J. Boehm, R. Cartwright, M. Riggle, M. J. O'Donnell,* Exact Real Arithmetic : A Case Study in Higher Order Programming*, Proc. 1986 ACM conf. LISP and functional programming, pages 162–173, 1986), où le test à* 0 *(utilisé implicitement dans le calcul* $\mathtt{A[k,k]} > \mathtt{0}$*) est indécidable.*

8 Distances minimales à sommet de départ fixé

Le problème des distances minimales entre *tous* les couples de sommets est résolu par l'algorithme de Floyd (Roy-Warshall généralisé) en temps $O(n^3)$. Si l'on souhaite trouver toutes les distances minimales à partir d'un seul sommet fixé, on va voir qu'il existe de nombreux algorithmes plus efficaces.

Dans cette section, on s'intéressera à des graphes valués $G \stackrel{\text{def}}{=} (S, A, c)$, où $c\colon A \to \mathbb{R}$. La valeur $c(x, y)$ est le *coût* de l'arc $x \to y$. C'est la valeur de l'arc dans le graphe valué G, mais la notation et le nom sont une indication du fait que nous chercherons à minimiser ces valeurs. De même, le poids d'un chemin sera appelé son coût, dans ce contexte.

Définition 8.1 *Le* coût $c(x \to^* y)$ *d'un chemin est la somme des coûts de ses arcs. La* distance $d(x, y)$ *d'un sommet x à un sommet y est la borne inférieure de l'ensemble des coûts des chemins de x à y.*

coût

distance

Donc, dans le cas où G n'a pas de circuit de coût strictement négatif, $d(x, y) = +\infty$ s'il n'y a pas de chemin de x à y dans le graphe, et sinon, c'est le coût d'un chemin de coût minimum de x à y, par la proposition 7.25.

On commence par faire une remarque facile, que nous utiliserons plus tard.

Proposition 2.2 (BBC, section 7.2.2, page 221) *Si $\gamma\colon x = x_0 \to x_1 \to \cdots \to x_n = y$ est un chemin de coût minimum dans un graphe valué G à valeurs dans $\mathbb{R}$, alors pour tous indices i, j tels que $0 \leq i \leq j \leq n$, $x_i \to \cdots \to x_j$ est un chemin de coût minimum de x_i à x_j dans G.*

Démonstration. Sinon, il existerait un chemin de x_i à x_j de coût strictement inférieur, et en l'installant à la place du sous-chemin de x_i à x_j dans γ, on obtiendrait un chemin de x à y de coût strictement inférieur à celui de γ. □

8.1 L'algorithme de Bellman-Ford

Définition 8.2 *Soit G un graphe valué à valeurs dans $\mathbb{R}$, et s un sommet fixé de G. Pour tout $k \in \mathbb{N}$, pour tout sommet x de G, on note $d_k(s, x)$ la borne inférieure des coûts des chemins de s à x de longueur au plus k dans G.*

La notation $d_k(s, x)$ ne fait pas apparaître ni G. Ceci a pour but la légèreté de notation, et se comprend dans la mesure où nous supposerons G fixé dans la suite.

Remarque 8.3 *Attention, dans $d_k(s, x)$, k est bien une* longueur*, et pas un* coût*. Il faudra bien faire attention à différencier les deux notions.*

La quantité $d_k(s, x)$ vaut $+\infty$ s'il n'y a pas de chemin de s à x de longueur au plus k, autrement s'il n'y a pas de chemin du tout de s à x, ou bien si tous les chemins de s à x sont de longueur strictement supérieure à k.

Sinon, $d_k(s,x)$ est le coût d'un *chemin de longueur $\leq k$ de coût minimum* (de s à x). Cette expression signifie que, parmi tous les chemins de longueur $\leq k$ (de s à x), on choisir celui de coût le plus petit. Attention, ce n'est pas la même chose qu'un chemin de coût minimum de longueur $\leq k$, ce qui signifierait un chemin de coût minimum (de s à x), à condition qu'il soit de longueur $\leq k$.

chemin de longueur $\leq k$ de coût minimum

Ceci permet de donner une récurrence simple permettant de calculer $d_k(s,x)$ pour tout sommet x et pour tout $k \in \mathbb{N}$.

Lemme 8.4 *Soit $G \stackrel{def}{=} (S, A, c)$ un graphe valué à valeurs dans $\mathbb{R}$. On a :*

1. *$d_0(s,x) = 0$ si $s = x$, $+\infty$ sinon ;*
2. *pour tout $k \in \mathbb{N}$,*

$$d_{k+1}(s,x) = \min(d_k(s,x), \min\{d_k(s,y) + c(y,x) \mid y \in S \text{ tel que } (y,x) \in A\}).$$

Démonstration. 1. Il y a un unique chemin de longueur 0 de s à s, et il est de coût nul. Il n'y a aucun chemin de longueur 0 de s à un sommet distinct de s.

2. On a d'abord :

$$d_{k+1}(s,x) \leq \min(d_k(s,x), \min\{d_k(s,y) + c(y,x) \mid y \in S \text{ tel que } (y,x) \in A\})$$

car tout chemin $s \to^* x$ de longueur $\leq k$ (compté dans $d_k(s,x)$) et tout chemin $s \to^* y \to x$ où $s \to^* y$ est de longueur $\leq k$ (compté dans le second min) est un chemin de s à x de longueur $\leq k+1$. (Cet argument est valable même s'il n'y a aucun de ces chemins : une quantification universelle n'a pas besoin, pour être appliquée, de porter sur un ensemble non vide.)

Pour démontrer l'inégalité inverse, il suffit de montrer que pour tout chemin γ: $s \to^* x$ de longueur $\leq k+1$ et de coût a, il existe soit : (A) un chemin de s à x de longueur $\leq k$ de coût $\leq a$, soit : (B) un chemin de s à y de longueur $\leq k$, pour un certain prédécesseur y de x dans G, et dont le coût b serait tel que $b + c(y,x) \leq a$. Si le chemin γ est de longueur $\leq k$, alors on a trouvé un chemin de type (A). Sinon, γ est un chemin non vide, qui s'écrit donc $s \to^* y \to x$ pour un certain sommet y. Soit b le coût du chemin $s \to^* y$ ainsi obtenu. Alors le coût de γ est $b + c(y,x)$, et ceci remplit les conditions de (B). □

On en déduit l'algorithme de Bellman-Ford de la table 13. L'algorithme est dû, comme son nom l'indique en partie, indépendamment à Bellman (R. Bellman, *On a Routing Problem*, Quart. Appl. Math. 16 :87–90, 1958) et à Ford (L. R. Ford, Jr., *Network Flow Theory*, Paper P-923, RAND Corporation, 1956). Il s'agit d'une implémentation quasiment directe des équations de Bellman, calculant dans le tableau `d`$[x,k]$ les valeurs de $d_k(s,x)$. À la fin de la boucle sur `k` (lignes 4–11), on dispose donc notamment de $d_n(s,x)$. La notation `G as (S, succ, c)` signifie que la variable `G` va contenir un triplet, et que ses trois composantes vont aussi se retrouver dans les variables `S`, `succ` et `c` ; il s'agit d'un morceau de syntaxe emprunté à Standard ML.

```
   fun bellman-1 (G as (S, succ, c), s, n) =
1      for each x ∈ S do
2              d[x,0] := +∞;
3      d[s,0] := 0;
4      for k=1..n do
5          for each x ∈ S do
6                  d[x,k] := d[x,k-1];
7          for each y ∈ S do
8                  for each x ∈ succ(y) do
9                          let improve = d[y,k-1]+c(y,x) in
10                         if improve<d[x,k]
11                            then d[x,k] := improve;
12     return d;
```

TABLE 13 – L'algorithme de Bellman-Ford, première version

Lemme 8.5 *Si G est une graphe valué, à valeurs dans $\mathbb{R}$, et à n sommets, pour tout sommet s de G, si G n'a pas de circuit de coût strictement négatif (et plus précisément, aucun circuit de coût strictement négatif accessible depuis s), alors pour tout sommet x de G, $d(s,x) = d_n(s,x)$.*

Démonstration. On a clairement $d(s,x) \leq d_n(s,x)$. Dans l'autre sens, pour tout chemin $\gamma\colon s \to^* x$ de longueur $> n$ et de coût c, il existe nécessairement un sommet répété sur ce chemin, qui est donc de la forme $s \to^* y \to^+ y \to^* x$. Le circuit $y \to^+ y$ étant de coût positif ou nul, le fait de l'enlever produit un chemin $s \to^* y \to^* x$ de longueur strictement plus petite et de coût inférieur ou égal. En itérant le procédé, on finira par obtenir un chemin de s à x de longueur $\leq n$ et de coût inférieur ou égal à c. En prenant les bornes inférieures sur tous les chemins γ, on obtient donc que $d(s,x) \geq d_n(s,x)$.

Théorème 8.6 *Si G est une graphe valué, à valeurs dans $\mathbb{R}$, et à n sommets, pour tout sommet s de G, si G n'a pas de circuit de coût strictement négatif (et plus précisément aucun circuit de coût strictement négatif accessible depuis s), alors* `bellman-1`(G,s,n) *calcule un tableau* `d` *tel que, pour tout sommet x de G,* `d`$[x,n]$ *vaut la distance $d(s,x)$. La complexité de cet algorithme est en $O(n(m+n))$.*

Pour ce qui est de la complexité, nous supposons que la complexité des opérations de base sur les réels est en temps constant, voir la remarque 7.26.

Démonstration. La correction vient des équations de Bellman (lemme 8.4) et du lemme 8.5. Pour ce qui est de la complexité, chaque opération élémentaire étant en temps constant, on réalise que le nombre de tours dans la boucle interne des lignes 9–11, pour un `k` fixé, c'est-à-dire lorsque `y` varie dans S et `x` varie parmi les successeurs de `y`, est de $\sum_{y \in S} \#\texttt{succ}(y) = m$. On doit y ajouter le temps

```
   fun bellman-2 (G as (S, succ, c), s, n) =
1      for each x ∈ S do
2              d[x] := +∞;
3      d[s] := 0;
4      for k=1..n do
5          for each y ∈ S do
6                  for each x ∈ succ(y) do
7                          let improve = d[y]+c(y,x) in
8                          if improve<d[x]
9                             then d[x] := improve;
10     return d;
```

TABLE 14 – L'algorithme de Bellman-Ford, calcul en place

constant d'initialisation de la boucle sur `x` (lignes 8–11) pour chaque sommet `y`, ce qui fait un temps supplémentaire proportionnel à n. Donc le temps pris par les lignes 5–11, pour chaque valeur de `k`, est proportionnel à $m + n$. On effectue n tours sur `k`, pour une complexité en $O(n(m + n))$, ce à quoi on doit ajouter le temps $O(n)$ de l'initialisation (lignes 1–3), qui est négligeable. □

On peut aussi coder l'algorithme de Bellman-Ford *en place*, voir la table 14. Ceci permet en plus d'éviter l'initialisation des lignes 5–6 de l'algorithme de la table 13. La complexité théorique reste la même, ceci ne faisant qu'économiser un temps $O(n^2)$. L'invariant nécessaire à la ligne 5) pour prouver la correction de cet algorithme est :

1. pour tout sommet x de G, $\mathtt{d}[x] = d_{\mathtt{k}-1}(\mathtt{s}, x)$.

Pour établir cet invariant, on a besoin d'établir un invariant plus complexe à la ligne 6. En supposant que le parcours des sommets de S à la ligne 6 s'effectue dans l'ordre $x_1, \ldots, x_n$ (ces sommets étant distincts deux à deux), cet invariant est la conjonction de :

2. il existe un unique indice i tel que $\mathtt{y} = x_i$;
3. pour tout sommet x, $\mathtt{d}[x]$ est le plus petit parmi les nombres $d_{k-1}(\mathtt{s}, x)$, $d_{k-1}(\mathtt{s}, x_j) + c(x_j, x)$ lorsque j parcourt les indices de $\{1, \cdots, i-1\}$ tels qu'il y a un arc $x_j \to x$.

On en déduit le résultat suivant.

Théorème 8.7 *Si G est une graphe valué, à valeurs dans $\mathbb{R}$, et à n sommets, et si G n'a pas de circuit de coût strictement négatif, alors pour tout sommet s de G,* `bellman-2`(G, s, n) *calcule un tableau* `d` *tel que, pour tout sommet x de G,* $\mathtt{d}[x]$ *vaut la distance $d(s, x)$. La complexité de cet algorithme est en $O(n(m+n))$.*

Nous supposons comme d'habitude que la complexité des opérations de base sur les réels est en temps constant ; voir la remarque 7.26.

```
   fun bellman-1-detect (G as (S, succ, c), s, n) =
1      for each x ∈ S do
2              d[x,0] := +∞;
3      d[s,0] := 0;
4      for k=1..n+1 do
5          for each x ∈ S do
6                  d[x,k] := d[x,k-1];
7          for each y ∈ S do
8                  for each x ∈ succ(y) do
9                          let improve = d[y,k-1]+c(y,x) in
10                         if improve<d[x,k]
11                            then d[x,k] := improve;
12     for each y ∈ S do
13             if d[y,n+1]<d[y,n]
14                then raise CircuitNegatif;
15     return d;
```

TABLE 15 – L'algorithme de Bellman-Ford, avec détection de circuits de coût négatifs

On peut aussi modifier l'une ou l'autre des versions de l'algorithme de Bellman-Ford pour *détecter* l'existence de circuits de coût strictement négatifs. L'algorithme de la table 15 a le même effet que `bellman-1` sur tout graphe sans circuit de coût négatif, et même sur tout graphe n'ayant aucun circuit de coût négatif accessible depuis `s`. Sur un graphe ayant un circuit de coût négatif accessible depuis `s`, il lance l'exception `CircuitNegatif`, avec comme argument un sommet `y` accessible depuis `s` tel qu'il existe un circuit $y \to^+ y$ de coût négatif. Ceci est dû au lemme suivant.

Lemme 8.8 *Soit G un graphe valué, à valeurs dans $\mathbb{R}$, et s un sommet de G. Il existe un circuit $z \to^+ z$ de coût négatif avec z accessible depuis s si et seulement si $d_{n+1}(s,y) < d_n(y)$ pour au moins un sommet y de G.*

Démonstration. Si $d_{n+1}(s,y) < d_n(s,y)$ pour un certain sommet y de G, alors il existe un chemin $\gamma\colon s \to^* y$ de longueur $\leq n+1$ de coût strictement inférieur à celui de tout chemin de s à x de longueur $\leq n$. En particular, γ est de longueur exactement $n+1$, et contient donc un sommet répété, disons z. On écrit γ sous la forme $s \to^* z \to^+ z \to^* y$. Si le circuit $z \to^+ z$ était de coût positif ou nul, le supprimer de γ fournirait un chemin $s \to^* z \to^* y$ de longueur $\leq n$ et coût inférieur ou égal à celui de γ, ce qui est impossible.

Sinon, alors $d_{n+1}(s,y) = d_n(s,y)$ pour tout sommet y de G, car l'inégalité $d_{n+1}(s,y) \leq d_n(s,y)$ est toujours vraie. Donc $d_{n+1} = d_n$. En utilisant le point 2 du

lemme 8.4, on en déduit $d_{n+2} = d_{n+1}$ pour tout sommet y de G, puis $d_{n+3} = d_{n+2}$, et ainsi de suite. Une récurrence immédiate montre donc que $d_m = d_n$ pour tout $m \geq n$.

S'il existait un circuit $z \to^+ z$ de coût $a < 0$, avec z accessible depuis s, alors soit k sa longueur, et soit j la longueur d'un chemin élémentaire γ de s à z, qu'on peut prendre de coût minimal parmi les chemins de s à z de longueur inférieure ou égal à n, c'est-à-dire de coût $d_n(z)$. Le chemin obtenu en concaténant à γ le circuit $z \to^+ z$ a un coût de $d_n(z) + a < d_n(z)$. Or, par définition, $d_{n+k}(z)$ is inférieur ou égal au coût de ce chemin, donc $d_{n+k}(z) < d_n(z)$. Ceci est impossible car $d_m = d_n$ pour tout $m \geq n$. □

8.2 Arbres de chemins de coûts minimums

Proposition 2.3 (BBC, section 7.2.2, page 222) *Pour tout graphe valué G à valeurs dans $\mathbb{R}$, sans circuit de coût strictement négatif, pour tout sommet s de G, il existe un sous-arbre $T \stackrel{def}{=} (\mathrm{Reach}(s), s, p)$ de G, de racine s, dont les sommets sont les sommets accessibles depuis s dans G, et tel que pour tout $x \in \mathrm{Reach}(s)$, l'unique chemin :*

$$s \to_T \cdots \to_T x$$

dans T est un chemin de coût minimum de s à x dans G.

*On appelle T l'*arbre des chemins de coûts minimums *de G.*

arbre des chemins de coûts minimums

Démonstration. Pour chaque sommet $x \in \mathrm{Reach}(s)$, il existe un chemin $\gamma_x \colon s \to^* x$ de coût minimum, par la proposition 7.25. Parmi ces chemins de coût minimum de s à x, on choisit de plus γ_x de *longueur minimum*. Soit $r(x)$ la longueur de γ_x.

Si $x \neq s$, γ_x est de la forme $s \to^* y \to x$. Par la proposition 2.2, le chemin $\gamma \colon s \to^* y$ obtenu comme préfixe de γ_x est de coût minimum de s à y, donc de coût égal à $c(\gamma_y)$. Le chemin γ_y n'est pas forcément identique au chemin γ, mais comme parmi tous les chemins de coût minimum de s à y, γ_y est de longueur minimum, sa longueur $r(y)$ est inférieure ou égale à celle de γ, qui vaut $r(x) - 1$.

Pour tout $x \in \mathrm{Reach}(s) \setminus \{s\}$, on pose $p(x) \stackrel{\text{def}}{=} y$. Ceci définit une fonction p. Nous venons de démontrer que $r(p(x)) < r(x)$ pour tout $x \in \mathrm{Reach}(s) \setminus \{s\}$. De la sorte, par récurrence sur $r(x)$, il existe nécessairement un entier n tel que $p^n(x) = s$, et ceci démontre que $T \stackrel{\text{def}}{=} (\mathrm{Reach}(s), s, p)$ est un arbre.

Montrons maintenant que l'unique chemin $s \to_T^* x$ dans l'arbre T est un chemin de coût minimum dans G, par récurrence sur $r(x)$.

Si $r(x) = 0$, c'est que $x = s$. Les chemins non triviaux de s à x sont donc des circuits, et par hypothèse leur coût est positif ou nul, donc supérieur ou égal au coût du chemin vide de s à x. Il s'ensuite que $d(s, x) = 0$. C'est aussi le coût de l'unique chemin (vide) $s \to_T^* x$ dans T.

Si $r(x) > 0$, on a vu que γ_x s'écrivait $s \to^* y \to x$, avec $y \stackrel{\text{def}}{=} p(x)$, et donc $y \to_T x$. De plus, $r(y) < r(x)$, ce qui permettra d'appliquer l'hypothèse de récurrence sur y. Comme on l'a dit plus haut, le chemin $\gamma\colon s \to^* y$ obtenu comme préfixe de γ_x n'est pas nécessairement égal à γ_y, mais on a vu qu'il avait le même coût que γ_y. Or, par hypothèse de récurrence, l'unique chemin $s \to_T^* y$ dans l'arbre T est un chemin de coût minimum de s à y dans G. Son coût est donc égal au coût de γ_y, et donc à celui de γ. Le coût de $s \to_T^* y \to_T x$ est donc égal au coût de γ plus $c(y, x)$, qui est par définition le coût de γ_x, c'est-à-dire le coût minimum de s à x dans G. □

Les algorithmes qui suivent vont résoudre la question du calcul des distances en construisant un arbre de chemins de coûts minimums, partant de l'arbre trivial $\{s\}$, et en ajoutant des arcs à l'arbre au fur et à mesure. Contrairement à l'algorithme `reach_set`, cet ajout d'arcs s'accompagnera parfois de la destruction d'arcs anciennement ajoutés.

En attendant, si l'on a produit un arbre (S_T, s, p_T), comment peut-on décider s'il est un arbre de chemins de coûts minimums ? C'est la question que résout la proposition suivante.

Proposition 2.4 (BBC, section 7.2.2, page 223) *Soit $G \stackrel{def}{=} (S, A, c)$ un graphe valué G à valeurs dans $\mathbb{R}$ et soit s un sommet de G. Soit $T \stackrel{def}{=} (S_T, s, p_T)$ un arbre avec $s \in S_T \subseteq \mathrm{Reach}(s)$. Pour tout $x \in S_T$, on définit $\lambda_T(x)$ comme étant le coût de l'unique chemin de s à x dans T ; on définit $\lambda_T(x)$ comme valant $+\infty$ pour tout autre sommet x de G.*

Alors T est un arbre de chemins de coûts minimums de G si et seulement si, pour tout sommet $x \in \mathrm{Reach}(s)$ et pour tout successeur y de x dans G,

$$\lambda_T(x) + c(x, y) \geq \lambda_T(y).$$

Démonstration. Si T est un arbre de chemins de coûts minimums, alors $\lambda_T(x) + c(x, y)$ est le coût d'un chemin $s \to_T^* x \to y$, qui est donc supérieur ou égal au coût minimum $d(s, y)$, lequel est égal à $\lambda_T(y)$ par hypothèse.

Réciproquement, pour tout chemin $s = x_0 \to x_1 \to \cdots \to x_n = x$ dans G, son coût est $\sum_{i=1}^{n} c(x_{i-1}, x_i)$. Si la condition de la proposition est vérifiée, cette somme est supérieure ou égale à $\sum_{i=1}^{n} (\lambda_T(x_i) - \lambda_T(x_{i-1})) = \lambda_T(x_n) - \lambda_T(x_0) = \lambda_T(x) - \lambda_T(s)$. Or $\lambda_T(s) = 0$ par définition. Donc tout chemin de s à x dans G a un coût supérieur ou égal à $\lambda_T(x)$. On en conclut que l'unique chemin $s \to_T^* x$, qui est aussi un chemin dans G, et donc le coût est $\lambda_T(x)$ par définition, est un chemin de coût minimum. □

8.3 L'algorithme de Ford (ou Ford, Gallo et Pallottino)

L'algorithme de Ford (L. R. Ford, Jr., *Network Flow Theory*, Paper P-923, RAND Corporation, 1956), dont l'explication a été simplifiée plus tard par Gallo

```
   fun ford (G as (S, succ, c), s, n) =
1      for each x ∈ S do
2               d[x] := +∞;
3               p[x] := undef;
4      d[s] := 0;
5      work := empty;
6      push(work,s);
7      while nonempty(work) do
8            let x=pop(work) in
9                for each y ∈ succ(x) do
10                        let improve = d[x]+c(x,y) in
11                        if improve<d[y]
12                           then d[y] := improve;
13                                p[y] := x;
14                                push_opt(work,y);
15     return d, p;
```

TABLE 16 – L'algorithme de Ford

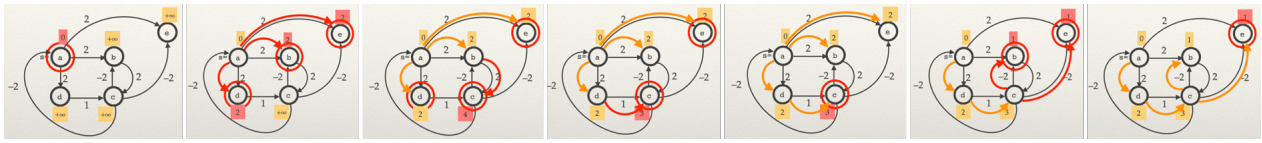


FIGURE 3 – Un exemple d'exécution de l'algorithme de Ford

et Pallottino (G. Gallo, S. Pallottino, *Shortest Path Methods : A Unifying Approach*, Math. Prog. Study 26 :38–64, 1986), est décrit en table 16. On a pris le parti d'une structure de travail abstraite, comme en section 4.6 (table 3), ce qui nous permettra de choisir librement plus tard si nous préférons la réaliser sous forme de pile, de file, ou de file à priorités. La fonction `push_opt` est une version optimisée de `push`, qui sur `work` et `y` appelle `push(work,y)` uniquement si `y` n'est pas déjà présent dans la liste `work`. Ce dernier test s'effectue en maintenant un tableau de bits, comme avec le tableau `onStk` qui nous avait servi à implémenter la pile `stk` dans l'algorithme de Tarjan en section 6.4.

Un exemple d'exécution de l'algorithme de Ford est donné à la figure 3, de gauche à droite. Les arcs sont étiquetés avec leurs coûts en noir. À tout moment, la valeur `d`$[y]$ de chaque sommet y est notée sur fond orange à côté du sommet, ou sur fond rouge si elle vient d'être mise à jour (donc diminuée). Les sommets entourés en rouge sont ceux de la structure de travail abstraite `work`.

L'exemple montre notamment qu'un sommet, une fois extrait de `work`, peut y être ajouté plus tard, contrairement à ce qui arrivait dans les algorithmes vus jusqu'ici : le sommet `b` est ajouté à la deuxième étape, enlevé à la troisième, et

rajouté de nouveau à la sixième.

Les invariants à maintenir en début de ligne 8 sont nombreux. Les voici.

(Inv 1) Le tableau `p` (vu comme fonction, et en interprétant la valeur `undef` comme indéfinie) défini un arbre $T \stackrel{\text{def}}{=} (S_T, \mathtt{s}, \mathtt{p})$ de racine `s`, avec $\mathtt{s} \in S_T \subseteq \text{Reach}(\mathtt{s})$, $S_T \stackrel{\text{def}}{=} \{x \in S \mid \mathtt{p}(x) \neq \mathtt{undef}\} \cup \{\mathtt{s}\}$.

(Inv 2) $\mathtt{d}[\mathtt{s}] = 0$.

(Inv 3) Pour tout sommet x de G hors de T (les sommets *libres*), $\mathtt{d}[x] = +\infty$. libres

(Inv 4) Pour tout arc $x \to_T y$ dans T, $\mathtt{d}[y] = \mathtt{d}[x] + c(x, y)$.

(Inv 5) $\mathtt{work} \subseteq S_T$.

(Inv 6) Pour tout sommet x dans $S_T \setminus \mathtt{work}$ (les sommets dits *fermés*, à ne pas confondre avec la notion de même nom de la définition 5.11), les successeurs de x dans G sont tous dans S_T. fermés

(Inv 7) Pour tout sommet fermé x, pour tout successeur y de x dans G, $\mathtt{d}[x] + \mathtt{c}(x, y) \geq \mathtt{d}[y]$.

On appellera finalement sommets *ouverts* les sommets de `work`. Nous utiliserons rarement les appellations « fermé » et « ouvert », qui ont l'inconvénient de varier au cours du temps, et mènent à des erreurs dans les raisonnements formels. ouverts

Voici quelques éléments permettant de démontrer la préservation de ces invariants. Malgré l'apparence relativement formelle des arguments, pour obtenir une réelle démonstration formelle, on doit encore spécifier d'autres invariants, au début de la ligne 10. Il s'agit de **(Inv 1)** à **(Inv 4)**, plus :

(Inv 5') $\mathtt{work} \subseteq S_T$ et $\mathtt{x} \in S_T$;

(Inv 6') les successeurs de tout sommet de $S_T \setminus (\mathtt{work} \cup \{\mathtt{x}\})$ sont dans S_T, et si l'énumération des successeurs de `x` à la ligne 9 s'effectue dans l'ordre y_1, …, y_k, alors $\mathtt{y} = y_i$ pour un unique index i, et y_1, …, y_{i-1} sont dans S_T ;

(Inv 7') pour tout sommet x de $S_T \setminus (\mathtt{work} \cup \{\mathtt{x}\})$, pour tout successeur y de x dans G, $\mathtt{d}[x] + \mathtt{c}(x, y) \geq \mathtt{d}[y]$, et $\mathtt{d}[\mathtt{x}] + \mathtt{c}(\mathtt{x}, y_1) \geq \mathtt{d}[y_1]$, …, $\mathtt{d}[\mathtt{x}] + \mathtt{c}(\mathtt{x}, y_{i-1}) \geq \mathtt{d}[y_{i-1}]$ (où y_1, …, y_k sont comme ci-dessus, et $\mathtt{y} = y_i$).

D'abord, on note que **(Inv 4)** et **(Inv 2)** impliquent :

(Inv42) le coût de l'unique chemin $\mathtt{s} \to_T^* x$ vers chaque sommet x de T est égal à $\mathtt{d}[x]$.

Pour un sommet `x` obtenu à la ligne 8 et un de ses successeurs `y` obtenu à la ligne 9, posons w, d, p, T les valeurs de `work`, `d`, `p` et l'arbre T défini par **(Inv 1)** au début de la ligne 10, et w', d', p', T' les valeurs des mêmes variables à la fin de la ligne 14. On va montrer que si **(Inv 1)**–**(Inv 4)** et **(Inv 5')**–**(Inv 7')** sont vrais au début de la ligne 8, ils sont de nouveau vrais à la fin de ligne 14. On pose $imp \stackrel{\text{def}}{=} d[\mathtt{x}] + \mathtt{c}(\mathtt{x}, \mathtt{y})$, la valeur de la variable `improve` calculée à la ligne 10.

— **(Inv 1)**. Si $imp \geq d[\mathtt{y}]$, alors $p' = p$, donc $T' = T$, et il n'y a rien à prouver. Supposons donc $imp < d[\mathtt{y}]$. La difficulté est de montrer que T', modifié à la ligne 13, reste un arbre, autrement dit que pour tout sommet z de T', il existe un entier n tel que $p'^n(z) = \mathtt{s}$. Si ce n'était pas le cas, comme la seule valeur qui change entre p et p' est celle en $\mathtt{y}$, et que $p'[\mathtt{y}] = \mathtt{x}$, c'est que l'on aurait $p^n[\mathtt{x}] = \mathtt{y}$ pour un certain entier n. Autrement dit, l'unique chemin de $\mathtt{s}$ à $\mathtt{x}$ dans T est de la forme $\mathtt{s} \to_T^* \mathtt{y} \to_T^* \mathtt{x}$. Par **(Inv42)**, le coût de ce chemin vaut $d[\mathtt{x}]$, et le coût de son préfixe $\mathtt{s} \to_T^* \mathtt{y}$ vaut $d[\mathtt{y}]$. Le coût du circuit $\mathtt{y} \to_T^* \mathtt{x} \to \mathtt{y}$ est alors de :

$$(d[\mathtt{x}] - d[\mathtt{y}]) + \mathtt{c}(\mathtt{x}, \mathtt{y}) = imp - d[\mathtt{y}]$$

qui est strictement négatif, puisque $imp < d[\mathtt{y}]$. Mais c'est impossible, car G n'a aucun circuit de coût strictement négatif. Donc T' est en fait un arbre.

— **(Inv 2)**. De nouveau, si $imp \geq d[\mathtt{y}]$, il n'y a rien à prouver. Supposons donc $imp < d[\mathtt{y}]$. Comme $\mathtt{d}[\mathtt{y}]$ est modifié à la ligne 12, pour être sûr que $d'(\mathtt{s}) = 0$, il suffit de vérifier que $\mathtt{y} \neq \mathtt{s}$. Si on avait $\mathtt{y} = \mathtt{s}$, on aurait un circuit $\mathtt{s} \to_T^* \mathtt{x} \to \mathtt{y} = \mathtt{s}$. Par **(Inv 42)**, son coût serait de $d[\mathtt{x}] + \mathtt{c}(\mathtt{x}, \mathtt{y}) = imp < d[\mathtt{y}] = d[\mathtt{s}]$ (car $\mathtt{y} = \mathtt{s}$) $= 0$ (par **(Inv2)**). C'est impossible, car il n'y a pas de circuit de coût strictement négatif dans G.

— **(Inv 3)**. Immédiat.

— **(Inv 4)**. Si $imp \geq d[\mathtt{y}]$, il n'y a rien à prouver car $T' = T$. Supposons donc $imp < d[\mathtt{y}]$. On a vu plus haut que T' était toujours un arbre. Les arcs de T' sont ceux de T, moins l'arc arrivant sur $\mathtt{y}$ s'il y en avait déjà un dans T, plus l'arc $\mathtt{x} \to_{T'}^* \mathtt{y}$. De plus, la seule différence entre d et d' est leur valeur en $\mathtt{y}$. Il suffit donc de montrer que $d'[\mathtt{y}] = d'[\mathtt{x}] + \mathtt{c}(\mathtt{x}, \mathtt{y})$.

Si $\mathtt{y} = \mathtt{x}$, l'arc $\mathtt{x} \to \mathtt{y}$ serait une boucle, donc un circuit. Son coût est $\mathtt{c}(\mathtt{x}, \mathtt{y}) = imp - d[\mathtt{x}] = imp - d[\mathtt{y}]$ (puisque $\mathtt{x} = \mathtt{y}$), qui serait strictement négatif puisque $imp < d[\mathtt{y}]$.

Comme ceci est impossible, on a $\mathtt{y} \neq \mathtt{x}$. Donc $d'[\mathtt{x}] = d[\mathtt{x}]$, puisque la ligne 12 ne modifie que les valeurs de $\mathtt{d}$ à l'indice $\mathtt{y}$. On en déduit que $d'[\mathtt{x}] + \mathtt{c}(\mathtt{x}, \mathtt{y}) = d[\mathtt{x}] + \mathtt{c}(\mathtt{x}, \mathtt{y}) = imp$, ce qui est égal à $d'[\mathtt{y}]$ par la ligne 12.

— **(Inv 5')**. Si $imp \geq d[\mathtt{y}]$, alors il n'y a rien à prouver. Sinon, $S_{T'} = S_T \cup \{\mathtt{y}\}$ par la ligne 13, et w' contient exactement les éléments de w plus $\mathtt{y}$ (ligne 14) — ce que l'on abrégera en $w' = w \cup \{\mathtt{y}\}$ —, donc, comme $w \subseteq S_T$ par **(Inv 5')**, on a $w' \subseteq S_{T'}$. De même, $\mathtt{x} \in S_{T'}$.

— **(Inv 6')**. On a $S_{T'} = S_T \cup \{\mathtt{y}\}$ et $w' = w \cup \{\mathtt{y}\}$, donc $S_{T'} \setminus (w' \cup \{\mathtt{x}\}) = (S_T \setminus (w \cup \{\mathtt{x}\})) \setminus \{\mathtt{y}\}$. Tout successeur d'un sommet de $S_{T'} \setminus (w' \cup \{\mathtt{x}\})$, qui est donc un successeur d'un sommet de $(S_T \setminus (w \cup \{\mathtt{x}\}))$, est dans S_T, donc dans $S_{T'}$. Pour ce qui est de $y_1, \ldots, y_{i-1}$, ils sont dans S_T par hypothèse, donc dans $S_{T'}$, et $y_i = \mathtt{y}$ est dans $S_{T'}$ par la ligne 13.

— **(Inv 7')**. Soit x un sommet de $S_{T'} \setminus (w' \cup \{\mathtt{x}\})$, c'est-à-dire de $(S_T \setminus (w \cup \{\mathtt{x}\})) \setminus \{\mathtt{y}\}$, comme on l'a vu plus haut. Pour tout successeur y de x dans G, on a $d(x) + \mathtt{c}(x, y) \geq d(y)$. De plus, $x \neq \mathtt{y}$, donc $d(x) = d'(x)$. Il s'ensuit que $d'(x) + \mathtt{c}(x, y) \geq d(y)$. On souhaite démontrer $d'(x) + \mathtt{c}(x, y) \geq d'(y)$, et pour ceci il suffit de vérifier que $d(y) \geq d'(y)$. C'est le cas si $d(y) = d'(y)$, ce qui arrive si $y \neq \mathtt{y}$ ou si $imp \geq d[\mathtt{y}]$. Si $y = \mathtt{y}$ et $imp < d[\mathtt{y}]$, alors $d'(y) = imp$ (ligne 12), qui est inférieur à $d[\mathtt{y}]$

  Pour chaque $j \in \{1, \cdots, i-1\}$, on a $d[\mathtt{x}] + \mathtt{c}(\mathtt{x}, y_j) \geq d[y_j]$. On a vu plus haut (en **(Inv 4)**) que $\mathtt{x} \neq \mathtt{y}$ si $imp < d[\mathtt{y}]$; sinon, $d = d'$. Dans les deux cas, $d[\mathtt{x}] = d'[\mathtt{x}]$. On souhaite montrer $d'[\mathtt{x}] + \mathtt{c}(\mathtt{x}, y_j) \geq d'[y_j]$, et pour ceci il suffit d'observer que $d[y_j] \geq d'[y_j]$, ce qui se fait comme on précédent paragraphe.

  Finalement, il ne reste qu'à montrer que $d'[\mathtt{x}] + \mathtt{c}(\mathtt{x}, y_i) \geq d'[y_i]$, c'est-à-dire que $d'[\mathtt{x}] + \mathtt{c}(\mathtt{x}, \mathtt{y}) \geq d'[\mathtt{y}]$. Si $imp \geq d[\mathtt{y}]$, alors $d = d'$, et ceci vient de la définition de $imp = d[\mathtt{x}] + \mathtt{c}(\mathtt{x}, \mathtt{y})$. Sinon, on a $d'[\mathtt{y}] = imp$ (ligne 12) $= d[\mathtt{x}] + \mathtt{c}(\mathtt{x}, \mathtt{y})$, et ceci est égal à $d'[\mathtt{x}] + \mathtt{c}(\mathtt{x}, \mathtt{y})$ puisque, comme on l'a vu en **(Inv 4)**, dans le cas $imp < d[\mathtt{y}]$, $\mathtt{x} \neq \mathtt{y}$.

On conclut.

Théorème 8.9 *Pour tout graphe valué G à valeurs dans $\mathbb{R}$ à n sommets, sans circuit de poids strictement négatif, pour tout sommet s de G, $\mathtt{ford}(G, s, n)$, s'il termine, retourne deux tableaux d et p tels que :*

1. *pour tout sommet $x \in \mathrm{Reach}(s)$, $d[x]$ est la distance $d(s, x)$ de s à x ;*
2. *l'arbre $T \stackrel{def}{=} (\mathrm{Reach}(s), s, p)$ est un arbre de chemins de coûts minimums.*

Démonstration. Si $\mathtt{ford}(G, s, n)$ termine et retourne d et p, alors, d'abord, `work` est vide. Par **(Inv 1)** et **(Inv 6)**, S_T contient `s` et est clos par l'opération successeur (dans G). Par le lemme 4.5, S_T contient donc $\mathrm{Reach}(s)$. Mais $S_T \subseteq \mathrm{Reach}(s)$, de nouveau par **(Inv 1)**, donc $S_T = \mathrm{Reach}(s)$. **(Inv 42)**, qui est une conséquence de **(Inv 4)** et de **(Inv 2)**, nous informe que $d[x] = \lambda_T(x)$ pour tout $x \in S_T$. Comme `work` est vide, tout sommet de S_T est fermé, et **(Inv 7)** nous dit alors que $\lambda_T(x) + \mathtt{c}(x, y) \geq \lambda_T(y)$ pour tout arc $x \to y$ de G. Par la proposition 2.4, T est donc un arbre de chemins de coûts minimums. Alors pour tout sommet x de G, $\lambda_T(x) = d(s, x)$, donc $d[x] = d(s, x)$. □

Nous nous intéressons maintenant à la terminaison et à la complexité de l'algorithme de Ford, en fonction des choix possibles d'implémentations de la structure de travail abstraite `work`.

8.4 L'algorithme de Ford à files

Si l'on réalise `work` via une file (ou FIFO, pour « first in first out »), alors on explore les sommets de $\mathrm{Reach}(s)$ en *largeur* (« breadth-first »). Il calcule alors

intuitivement d'abord tous les $d_0(x)$, puis tous les $d_1(x)$, etc., comme l'algorithme de Bellman-Ford.

Pour le démontrer, nous allons analyser à la place une variante de l'algorithme, et arguer du fait que la variante et l'algorithme calculent la même chose, et que le temps d'exécution de la variante est égal à celui de `ford` à peu de choses près. La variante est :

```
   fun ford-variant (G as (S, succ, c), s, n) =
1      for each x ∈ S do
2              d[x] := +∞;
3              p[x] := undef;
4      d[s] := 0;
5      work := empty;
6      push(work,s);
7      for k=1..n+1
8          work := ford-variant-phase (G, s, n, d, p, work, k);
9      return d, p;
```

où la fonction auxiliaire `ford-variant-phase` est définie comme suit :

```
   fun ford-variant-phase (G as (S, succ, c), s, n, d, p, work, k) =
   (* Pre I4(G,d,k−1) et k ≥ 1 et I5(G,work) *)
1      let later := empty in
2      while nonempty(work) do
3            let x=pop(work) in
4                for each y ∈ succ(x) do
5                        let improve = d[x]+c(x,y) in
6                        if improve<d[y]
7                           then d[y] := improve;
8                                p[y] := x;
9                                if y ∉ work then push_opt(later,y);
10     return later;
   (* Post I4(G,d,k) et I5(G,later) et work est vide *)
```

Le corps de `ford-variant-phase` est identique aux lignes 7–14 de l'algorithme de Ford (table 16), sauf que `y` est empilé sur une deuxième file `later`, au lieu de `work` ; le fait de le faire uniquement si `y` $\notin$ `work` permet de simuler exactement l'algorithme de Ford, du moins à condition de montrer que :

> (A) Lorsque l'on sort de la boucle sur `k` à la ligne 9 de `ford-variant`, alors `work` est vide.

L'autre différence est que `ford-variant-phase` est appelée au sein d'une boucle sur un indice `k`, souvent appelé la *phase* de l'algorithme, qui va de 1 à n (`n` dans le programme).

Il ne devrait pas être difficile de se convaincre que, à un terme additif près correspondant à la gestion de l'indice `k` (en $O(n)$: on incrémente `k` $O(n)$ fois), la complexité de `ford-variant` est égale à celle de `ford`, à condition de démontrer (A). Pour ceci, nous montrons que les invariants mentionnés dans les pré-conditions et les post-conditions de `ford-variant-phase` sont préservés. L'invariant I_4 énonce : pour tout sommet y de `G`, `d`$[y]$ est inférieur ou égal à $d_{\mathtt{k}}(s, y)$. L'invariant $I_5(\mathtt{G}, \mathtt{work})$ énonce que `work` est une liste de sommets de G, sans doublon.

Voyons rapidement pourquoi cet invariant est préservé. Réservons la lettre $\mathtt{d}_0$ pour la valeur du tableau `d` à l'entrée de la fonction `ford-variant-phase`, et $\mathtt{d}_1$ pour sa valeur à la sortie de `ford-variant-phase`. Le cas de l'invariant I_5 est évident, nous nous concentrons donc sur I_4. On suppose donc $I_4(\mathtt{G}, \mathtt{d}_0, \mathtt{k}-1)$, $\mathtt{k} \geq 1$, et nous démontrons $I_4(\mathtt{G}, \mathtt{d}_1, \mathtt{k})$.

Les valeurs des entrées de `d` ne peuvent que diminuer, et donc un invariant des lignes 1–10 de `ford-variant-phase` est que $\mathtt{d}_0[x] \geq \mathtt{d}[x] \geq \mathtt{d}_1[x]$ pour tout sommet x de `G`.

Pour tout sommet y de G, on souhaite démontrer que $\mathtt{d}_1[y] \leq d_{\mathtt{k}}(s, y)$. Or $\mathtt{d}_1[y]$ est calculé par `ford-variant-phase` comme étant le minimum de $\mathtt{d}_0[y]$ et de toutes les valeurs $\mathtt{d}[x] + c(x, y)$, lorsque x parcourt les prédécesseurs de y, et où `d` est la valeur courante du tableau de même nom aux lignes 5–6. Comme $\mathtt{d}_0[x] \geq \mathtt{d}[x]$, $\mathtt{d}_1[y] \leq \min(\mathtt{d}_0[y], \min\{\mathtt{d}[x] + c(x, y) \mid x \to y\})$. Grâce à la pré-condition, on sait que $\mathtt{d}_0[y] \leq d_{\mathtt{k}-1}(s, y)$, et que $\mathtt{d}_0[x] \leq d_{\mathtt{k}-1}(s, x)$. Donc $\mathtt{d}_1[y] \leq \min(d_{\mathtt{k}-1}(s, x), \min\{d_{\mathtt{k}-1}(s, y) + c(y, x) \mid x \to y\})$. Autrement dit, et en utilisant le point 2 du lemme 8.4, $\mathtt{d}_1[y] \leq d_{\mathtt{k}}(s, y)$.

La pré-condition de `ford-variant-phase` est vérifiée à son premier appel, où $k = 1$, car $d_0(s) = 0$ et $d_0(x) = +\infty$ pour tout sommet $x \neq s$, grâce au point 1 du lemme 8.4.

On en conclut enfin que (A) est vraie. En utilisant la post-condition de `ford-variant-phase` pour $\mathtt{k} = n$ ($= \mathtt{n}$, dans le code) c'est-à-dire à la fin de l'avant-dernière itération de la boucle sur `k` dans `ford-variant`, on a $\mathtt{d}[x] \leq d_n(s, x)$; et en l'absence de circuit de coût strictement négatif, $d_n(s, x) = d(s, x)$. Ceci implique que rappeler `ford-variant-phase` avec $\mathtt{k} = n + 1$, pour une dernière itération, ne peut mener à aucune amélioration du tableau `d` (ligne 7 de `ford-variant-phase`), ce qui implique que `later` restera vide lors de cette dernière exécution de `ford-variant-phase`. C'est ce qui est affecté à `work` en fin de `ford-variant` : nous avons démontré (A). (Bien entendu, ceci montre aussi que, en l'absence de circuit de coût négatif, il ne sert à rien d'effectuer cette dernière itération sur `k`, et que l'on pourrait donc faire varier `k` de 1 à `n` plutôt que $\mathtt{n} + 1$ à la ligne 7 de `ford-variant`.)

La complexité de `ford-variant-phase` est facile à analyser. Par la pré-condition I_5, la boucle des lignes 2–9 ne peut faire qu'on plus n tours, un au plus par sommet distinct `x`. Chaque tour de boucle prend un temps au plus $\#\mathtt{succ}(\mathtt{x})$ plus une constante, donc `ford-variant-phase` prend un temps qui est une fonc-

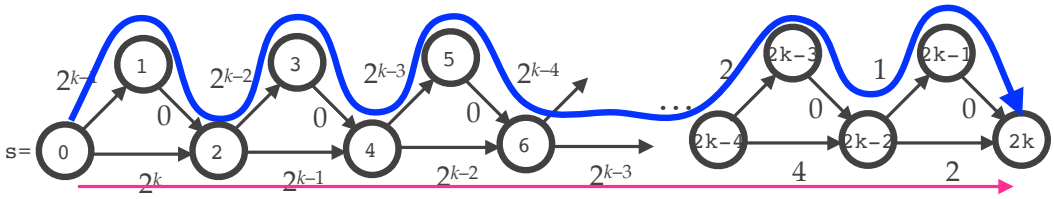


FIGURE 4 – Un exemple d'un graphe valué sur lequel l'algorithme de Ford à liste de travail prend un temps exponentiel

tion affine de $\sum_{x \in S}(\#\texttt{succ}(x) + 1) = n + m$. La fonction `ford-variant` appelle `ford-variant-phase` $n + 1$ fois, ce à quoi on doit ajouter le temps $O(n)$ de l'initialisation, pour un temps total en $O(n(n + m))$.

Comme la complexité de `ford` est égale à celle de `ford-variant` à un $O(n)$ près, on en déduit le théorème suivant.

Théorème 8.10 *La complexité de l'algorithme de Ford à files est en* $O(n(m+n))$.

Comme d'habitude, ce résultat suppose que la complexité des opérations de base sur les réels est en temps constant ; voir la remarque 7.26.

On notera aussi que `ford-variant` permet de détecter l'existence de circuits négatifs : il suffit de vérifier que la file `work` retournée au dernier appel de `ford-variant-phase` est non vide (cette fois-ci faire varier `k` de 1 à `n` + 1 et non `n` est important).

L'algorithme `ford-variant` ressemble énormément à l'algorithme de Bellman-Ford en place (table 14). La principale différence est qu'il examine uniquement les arcs $x \to y$ tels que x est extrait d'une file `work`, alors que l'algorithme de Bellman-Ford examine *tous* les sommets du graphe G (ligne 5 de la table 14). L'algorithme `ford-variant`, ou `ford` avec files, ne met à jour les distances que des sommets qui ont une chance de mener à une amélioration. Même si sa complexité dans le pire cas est la même que celle de l'algorithme de Bellman-Ford, en pratique il est bien meilleur.

8.5 L'algorithme de Ford à liste de travail

Si l'on implémente l'algorithme de Ford avec une *liste de travail* `work`, la complexité change du tout au tout. On peut montrer (G. Gallo, S. Pallottino, *Shortest Path Methods : A Unifying Approach*, Math. Prog. Study 26 :38–64, 1986) qu'elle passe alors en $\Theta(n2^n)$. Pour chacun des n sommets x, cette méthode peut avoir à mettre à jour $\texttt{d}[x]$ autant de fois qu'il y a de chemins élémentaires de s à x.

Un exemple exhibant ce comportement a été donné par Shier et Witzgall (D. R. Shier, Ch. Witzgall, *Properties of Labeling Methods for Determining Shortest*

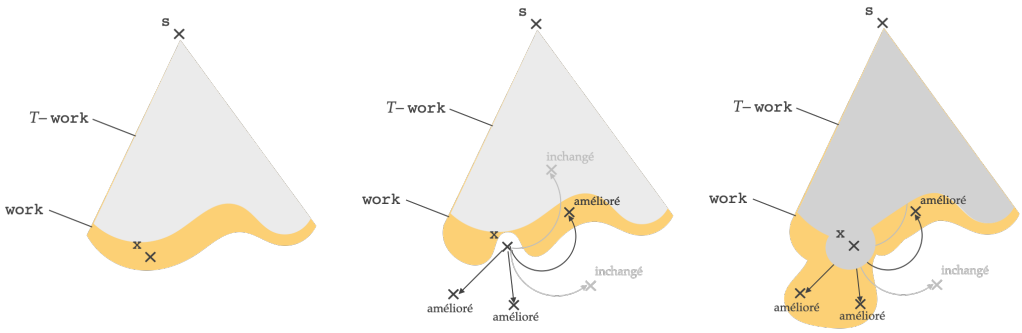


FIGURE 5 – Ce que fait l'algorithme de Dijkstra

Path Trees, J. Res. Nat. Bur. Standards 86(2), 1981, Section 5, p.326), et est affiché à la figure 4. Il est inspiré d'un exemple de Johnson portant sur l'algorithme de Dijkstra, que nous verrons à la section 8.6. Le sommet `s` est complètement à gauche, et les coûts des arcs sont affichés à côté des arcs concernés. Les arcs $2i+1 \to 2i+2$ ont un coût nul, les arcs $2i \to 2i+2$ ont un coût égal à 2^{k-i}, et les arcs $2i \to 2i+1$ ont un coût égal à 2^{k-i-1}.

L'arbre des chemins de coûts minimums n'a qu'un branche, représentée en bleu, et qui passe par les sommets `0`, `1`, `2`, ..., `2k`. On notera que le chemin le plus court (de longueur, et non de coût, le plus faible), est le chemin `0`, `2`, `4`, ..., `2k` du bas. On peut vérifier que, en supposant que les successeurs de chaque sommet pair $2i$ soient listés dans l'ordre $[2i+2, 2i+1]$ (ou de façon équivalente, en préférant explorer les chemins les plus courts d'abord), l'algorithme de Ford à liste de travail va effectuer $n2^n$ itérations avant de terminer.

8.6 L'algorithme de Dijkstra

L'algorithme de Dijkstra (E. W. Dijkstra, *A Note on Two Problems in Connexion with Graphs*, Numerische Mathematik 1 :269–271, 1959) était initialement conçu pour fonctionner sur des graphes valués, à valeurs dans $\mathbb{R}$, et sans *arc* de coût strictement négatif, ce qui est une hypothèse strictement plus forte que de ne pas avoir de *circuit* de coût strictement négatif.

Nous appelons algorithme de Dijkstra la variante de l'algorithme de Ford où `work` est implémenté par une file *à priorités*, la priorité de chaque sommet x étant la valeur courante `d`$[x]$. On remarquera que les valeurs des priorités changent au cours de l'algorithme, et l'on doit donc utiliser une implémentation de files à priorités avec mise à jour des priorités.

L'algorithme de Dijkstra, étant un cas particulier de l'algorithme de Ford, est correct (Theorem 8.9). Le point important est qu'il est plus rapide même que l'algorithme de Ford à files, dans le cas important où aucun *arc* n'est de coût strictement négatif.

L'explication semi-formelle en est simple. À tout moment de l'exécution de

l'algorithme de Ford, on se trouve dans une situation telle qu'en partie gauche de la figure 5. On verra que l'algorithme de Dijkstra vérifiera deux invariants particuliers : d'abord, pour tout sommet fermé y (dans $S_T \setminus \texttt{work}$, en gris), $\texttt{d}[y]$ est déjà optimale, c'est-à-dire égale au coût minimum $d(\texttt{s}, y)$; ensuite, tous les sommets ouverts, c'est-à-dire dans `work` (en orange, en bas), sont des feuilles de T. On sélectionne maintenant un sommet `x` de la file de travail `work`, et l'utilisation de la file de priorités garantit que $\texttt{d}[x]$ est minimum. Nous verrons que ceci implique que $\texttt{d}[x]$ est elle aussi optimale, du fait de ce choix, et du fait que G ne contient aucun arc de coût strictement négatif. On explore ensuite les successeurs `y` de `x`. Certains seront améliorés — c'est-à-dire que la valeur de $\texttt{d}[\texttt{y}]$ sera diminuée à la ligne 12 — et d'autres non. Comme tout sommet fermé est déjà optimal, aucun des successeurs `y` améliorés de `x` ne peut être fermé : voir la partie médiane de la figure 5. Seuls les sommets améliorés sont ajouté à `work` et à T, et l'on aboutit à la situation de la partie droite de la figure 5, où l'on a rétabli les deux invariants annoncés. Ceci montre aussi que chaque sommet `x` extrait de `work` (à la ligne 8) ne sera plus jamais réinstallé dans `work` : tout sommet fermé le reste à l'avenir, et ceci est une particularité de l'algorithme de Dijkstra (à condition qu'aucune arc de G n'ait un coût strictement négatif), que l'algorithme de Ford n'a pas en général. On en déduira la complexité de l'algorithme de Dijkstra.

Pour ceci, comme d'habitude, nous avons besoin d'invariants supplémentaires (au début de la ligne 8 de l'algorithme de la table 16). Le premier a déjà utilisé, les deux suivants sont nouveaux. Il est entendu que nous les prouverons sous les deux hypothèses que : (i) G n'a aucun arc de coût strictement négatif, et (ii) `work` est une file à priorités, de sorte que le sommet `x` obtenu à la ligne 8 est un sommet x tel que $\texttt{d}[x]$ est minimum (dans `work`).

(Inv 8) tout sommet apparaît au plus une fois dans `work` ;

(Inv 8D) pour tout sommet $y \in S_T \setminus \texttt{work}$, $\texttt{d}[y] = d(\texttt{s}, y)$;

(Inv 9D) tout sommet $y \in \texttt{work}$ est une feuille de T.

On a déjà vu que l'invariant **(Inv 8)** était préservé. Pour les autres, on est amené à démontrer les invariants suivants au début de la ligne 10.

(Inv 8Dx) $\texttt{d}[\texttt{x}] = d(\texttt{s}, \texttt{x})$;

(Inv 8D') pour tout sommet $y \in S_T \setminus (\texttt{work} \cup \{\texttt{x}\})$, $\texttt{d}[y] = d(\texttt{s}, y)$;

(Inv 9D) tout sommet $y \in \texttt{work}$ est une feuille de T.

C'est l'invariant **(Inv 8Dx)** qui est le plus important. En ligne 8, **(Inv 8D)** nous garantit que tout sommet $y \in S_T \setminus \texttt{work}$ avait une valeur $\texttt{d}[y]$ égale à la distance minimum $d(\texttt{s}, y)$, mais ne garantit rien du tout sur le sommet `x` qui, étant dans `work`, n'est justement pas dans $S_T \setminus \texttt{work}$.

Montrons pourquoi **(Inv 8Dx)** est nécessairement vrai juste après l'exécution de la ligne 8 (en supposant que les invariants usuels **(Inv 1)**–**(Inv 8)**, ainsi que **(Inv 8D)** et **(Inv 9D)** sont vrais juste avant). Si $\texttt{x} = \texttt{s}$, c'est clair : on a $\texttt{d}[\texttt{s}] = 0$ par **(Inv 2)**, et ceci vaut $d(\texttt{s}, \texttt{s})$, car comme il n'y a pas d'arc de coût strictement

négatif, il n'y a pas non plus de circuit de coût strictement négatif. On se place désormais dans le cas où $\mathtt{x} \neq \mathtt{s}$. Comme $\mathtt{x}$ était dans `work` au début de la ligne 8, il est dans S_T par **(Inv 5)**. Il y a donc un chemin $\mathtt{s} \to_T^* \mathtt{x}$ dans l'arbre T, donc dans G. Soit donc γ un chemin de coût minimum de $\mathtt{s}$ à $\mathtt{x}$ dans G. Soit aussi w la valeur de `work` avant l'exécution de la ligne 8, et w' sa valeur après. Donc $w = w' \cup \{\mathtt{x}\}$, et par **(Inv 8)**, $w' = w \setminus \{\mathtt{x}\}$. On observe que $\mathtt{s} \in S_T \setminus w$ et que $\mathtt{x} \notin S_T \setminus w$:

— $\mathtt{s} \in S_T$ par **(Inv 1)**, et si $\mathtt{s}$ était dans w, par **(Inv 9D)** il serait un feuille de l'arbre T. Mais il en est aussi la racine et si c'était le cas, S_T serait réduit au seul sommet $\mathtt{s}$. C'est impossible, car $\mathtt{x} \neq \mathtt{s}$, et $\mathtt{x} \in w \subseteq S_T$, cette dernière inclusion étant par **(Inv 5)**.

— $\mathtt{x} \notin S_T \setminus w$, car $\mathtt{x} \in w$.

Le chemin $\gamma \colon \mathtt{s} \to^* \mathtt{x}$ est donc un chemin allant d'un sommet de $S_T \setminus w$ à un sommet hors de $S_T \setminus w$. Il est donc de la forme $\mathtt{s} \to^* u \to v \to^* \mathtt{x}$, où u est le dernier sommet de γ dans $S_T \setminus w$. Notons que v n'est pas dans $S_T \setminus w$, et donc soit n'est pas dans S_T, soit est dans w. On a maintenant la chaîne d'inégalités suivantes :

$$
\begin{aligned}
d(\mathtt{s},\mathtt{x}) &= c(\gamma) && \gamma \text{ est un chemin de coût minimum} \\
&= c(\mathtt{s} \to^* u \to v) + c(v \to^* \mathtt{x}) \\
&\geq c(\mathtt{s} \to^* u \to v) && \text{car } c(v \to^* \mathtt{x}) \geq 0, \\
& && \text{aucun arc n'étant de } \textit{coût strictement négatif} \\
&= d(\mathtt{s},u) + \mathtt{c}(u,v) && \text{par la proposition 2.2} \\
&= \mathtt{d}[u] + \mathtt{c}(u,v) && \text{par } \textbf{(Inv 8D)}, \text{ puisque } u \in S_T \setminus w \\
&\geq \mathtt{d}[v] && \text{par } \textbf{(Inv 7)}, \text{ puisque } u \in S_T \setminus w \\
&\geq \mathtt{d}[\mathtt{x}] && \begin{cases} \text{si } v \notin S_T : & \mathtt{d}[v] = +\infty \\ \text{si } v \in w : & \mathtt{d}[\mathtt{x}] \leq \mathtt{d}[v], \text{ puisque} \\ & \mathtt{d}[\mathtt{x}] \text{ est } \textit{minimum} \text{ (dans } w) \end{cases} \\
&\geq d(\mathtt{s},\mathtt{x}) && \text{car } \mathtt{d}[\mathtt{x}] \text{ est le coût d'un chemin par } \textbf{(Inv 42)}.
\end{aligned}
$$

Ceci termine de démontrer que **(Inv 8Dx)** est vrai juste après l'exécution de la ligne 8, et donc à l'initialisation de la boucle des lignes 9–14. Le fait que **(Inv 8D')** et **(Inv 9D)** soient vrai à l'initialisation de cette boucle est une conséquence directe du fait que **(Inv 8D)** et **(Inv 9D)** étaient vraie avec l'extraction de $\mathtt{x}$ de `work` à la ligne 8.

Montrons maintenant que les invariants **(Inv 8Dx)**, **(Inv 8D')** et **(Inv 9D)** sont préservés par le corps de la boucle, c'est-à-dire par les lignes 10–14. C'est beaucoup moins intéressant, et nous le montrons par souci d'exactitude. Appelons w, d, p, T les valeurs de `work`, `d`, `p` et l'arbre T (défini à partir de `p` par **(Inv 1)**) au début de la ligne 10, et w', d', p', T' les valeurs des mêmes variables à la fin de la ligne 14. Posons $imp \stackrel{\text{def}}{=} d[\mathtt{x}] + \mathtt{c}(\mathtt{x},\mathtt{y})$ la valeur affectée à la variable `improve`

à la ligne 10. Si $imp \geq d[\mathtt{y}]$, alors $w' = w$, $d' = d$, $p' = p$ et donc $T' = T$, et les invariants **(Inv 8D)** et **(Inv 9D)** sont automatiquement préservés. Supposons donc que ce ne soit pas le cas. On a :

1. $imp < d[\mathtt{y}]$, comme on vient de le supposer ;
2. $\mathtt{x} \notin w$, puisque `x` a été enlevé de `work` à la ligne 8, et qu'aucun sommet n'apparaît plusieurs fois dans `work` par **(Inv 8)** ;
3. $d'[\mathtt{y}] = imp$, et $d'[z] = d[z]$ pour tout sommet $z \neq \mathtt{y}$, par la ligne 12 ;
4. $p'[\mathtt{y}] = \mathtt{x}$, et $p'[z] = p[z]$ pour tout sommet $z \neq \mathtt{y}$, par la ligne 13 ;
5. $w' = w \cup \{\mathtt{y}\}$, par la ligne 14 ;
6. $d[\mathbf{x}] = d(\mathtt{s}, \mathbf{x})$ (**(Inv 8Dx)**)
7. pour tout sommet $y \in S_T \setminus (w \cup \{\mathtt{x}\})$, $d[y] = d(\mathtt{s}, y)$ (**(Inv 8D')**) ;
8. tout sommet $y \in w$ est une feuille de T (**(Inv 9D)**).

On vérifie maintenant **(Inv 8D)**, **(Inv 8D')** et **(Inv 9D)**. On observe d'abord que $\mathtt{x} \neq \mathtt{y}$. En effet, par le point 1, $imp = d[\mathtt{x}] + \mathtt{c}(\mathtt{x}, \mathtt{y}) < d[\mathtt{y}]$. Si on avait $\mathtt{x} = \mathtt{y}$, ceci impliquerait $\mathtt{c}(\mathtt{x}, \mathtt{y})$, ce qui est impossible puisque G n'a aucun arc de coût strictement négatif.

(Inv 8D) On souhaite montrer que $d'[\mathbf{x}] = d(\mathtt{s}, \mathbf{x})$. Il suffit de s'apercevoir que $d'[\mathbf{x}] = d[\mathbf{x}]$, et d'utiliser le point 6 ci-dessus. Pour démontrer $d'[\mathbf{x}] = d[\mathbf{x}]$, on utilise le point 3, ce que l'on peut faire car $\mathtt{x} \neq \mathtt{y}$.

(Inv 8D') On souhaite montrer que pour tout sommet $y \in S_{T'} \setminus w'$, $d'[y] = d(\mathtt{s}, y)$. Par le point 4, $S_{T'} = S_T \cup \{\mathtt{y}\}$, et $w' = w \cup \{\mathtt{y}\}$ par le point 5. Donc $S_{T'} \setminus (w' \cup \{x\}) = S_T \setminus (w \cup \{\mathtt{x}, \mathtt{y}\}) \subseteq S_T \setminus (w \cup \{\mathtt{x}\})$. On utilise le point 7 (**(Inv 8D')**), et l'on obtient que $d[y] = d(\mathtt{s}, y)$. Il ne reste qu'à démontrer que $d'[y] = d[y]$, ce qui est vrai par le point 3. Ceci s'applique car, $y \neq \mathtt{y}$, vu que $y \in S_{T'} \setminus (w' \cup \{x\}) = S_T \setminus (w \cup \{\mathtt{x}, \mathtt{y}\})$.

(Inv 9D) On souhaite montrer que tout sommet $y \in w'$ est une feuille de T'. Par le point 5, $w' = w \cup \{\mathtt{y}\}$. Comme T' est un arbre par **(Inv 1)**, on déduit du point 4 que les feuilles de T' sont celles de T moins `x`, plus `y`. (Notamment, `y` est une feuille de T', sinon il y aurait un circuit dans T', ce qui est impossible puisque T' est un arbre.) Or tout élément de w est une feuille de T par le point 8 (**(Inv 9D)**), est différent de `x` par le point 2, et est donc une feuille de T' ; et `y` lui-même est une feuille de T', comme on vient de le voir.

Ceci termine la vérification des invariants. Considérons maintenant un sommet `x` quelconque extrait de `work` à la ligne 8 ; `x` est une feuille de l'arbre T avant l'exécution de la boucle des lignes 9–14, par **(Inv 9D)**. Cette boucle explore les $\# \succ (\mathtt{x})$ successeurs `y` de `x`, et ceux qui mènent à une amélioration de `d[y]` à la ligne 12 sont soit déjà dans `work`, soit hors de l'ensemble S_T de l'ensemble courant des sommets de T. En effet, sinon, `y` serait dans $T \setminus$ `work`, mais aucune amélioration n'est possible pour ces sommets, par **(Inv 8D)**.

Si l'on appelle *sommets à traiter* ceux de `work`, plus ceux hors de S_T, les lignes 13–14 ont donc pour effet de laisser les sommets à traiter inchangés : si `y` est déjà dans `work`, il le reste, et si `y` était hors de S_T, il rentre dans `work`. D'autre part, la ligne 8 élimine un sommet de `work`, qui était dans S_T par **(Inv 5)**. Ceci a pour effet de diminuer le nombre de sommets à traiter d'exactement 1.

On en conclut que la boucle des lignes 7–14 ne fait qu'au plus n tours. Chaque sommet `x` n'est extrait de `work` à la ligne 8 qu'au plus une fois lors de l'exécution de l'algorithme.

Théorème 8.11 *La complexité de l'algorithme de Dijkstra sur un graphe valué sans arc de coût strictement négatif, est en $O(m + n \log n)$, à condition d'implémenter* `work` *sous forme d'une liste de priorités via des tas de Fibonacci.*

Comme d'habitude, ce résultat suppose que la complexité des opérations de base sur les réels est en temps constant ; voir la remarque 7.26.

Démonstration. [Esquisse] Pour chaque sommet `x` extrait à la ligne 8, on explore ses $\# \succ (\texttt{x})$ successeurs. Si le temps d'exploration de chaque successeur `y` était constant, le temps d'exécution de l'algorithme de Dijkstra serait inférieur ou égal à quelque chose de proportionnel à $\sum_{x \in S} (\#\texttt{succ}(x) + 1) = m + n$.

Le coût de la manipulation de la liste `work` n'est pas constant, mais on peut s'en approcher. Pour ceci, on utilise le résultat suivant sur les tas de Fibonacci, vu en première partie de ce cours (pas dans ces notes). Pour un tas de Fibonacci τ, en notant $a(\tau)$ le nombre d'arbres du tas, $m(T)$ le nombre de sommets marqués, on peut définir un potentiel $pot(T) \stackrel{\text{def}}{=} C(a(T) + 2m(T))$, où C est une constante strictement positive appropriée. Alors l'insertion, l'union et la diminution de valeur (c'est pourquoi nous utilisons un tas de Fibonacci) sont en temps *amorti* constant, et l'opération `pop` s'effectue en temps *amorti* logarithmique. Le potentiel valant zéro au début et à la fin de la boucle des lignes 7–14 (de la table 16), puisque le potentiel d'un arbre vide est nul, le temps amorti coïncide avec le temps effectif. La boucle des lignes 7–14 prend donc un temps proportionnel à $n \log n$ (le temps total des opérations `pop` de la ligne 8, en notant que la taille de `work` ne peut dépasser n, les doublons en étant éliminés par **(Inv 8)**), plus $m + n$ (le temps total de l'exécution des lignes 9–14 au cours du déroulement de l'algorithme). Le total est donc bien en $O(m + n \log n)$.

À ceci, il faut en principe ajouter le temps de l'initialisation, aux lignes 1–6, qui est en $O(n)$, mais $O(m + n \log n) + O(n) = O(m + n \log n)$. □

La proposition 2.9 du BBC annonce un temps de la forme $O(m \log n)$. On peut noter que $O(m + n \log n)$ est meilleur si $n \leq m$, ce qui est un cas courant. Mais surtout, la proposition 2.9 du BBC est probablement erronée. Par exemple, un graphe à n sommets isolés (pas d'arc, donc $m = 0$) serait traité en temps nul si on la croit. Or, rien que l'initialisation aux lignes 1–6 prend un temps $O(n)$. On pourrait objecter qu'une notation O n'a de sens que lorsque *tous* les paramètres tendent vers $+\infty$, mais même avec cette interprétation, la proposition 2.9 du

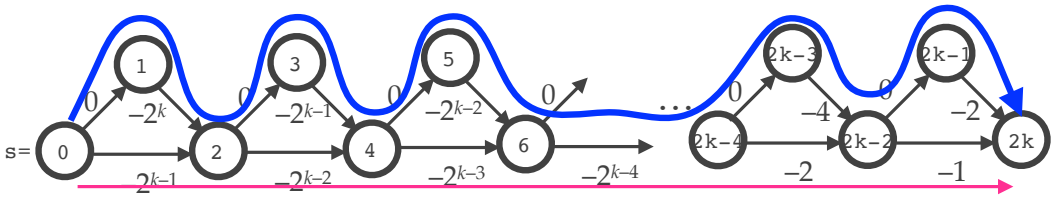


FIGURE 6 – Un exemple d'un graphe valué sur lequel l'algorithme de Dijkstra prend un temps exponentiel ; la contrainte « pas d'arc de coût strictement négatif » n'est pas respectée, mais il n'y a pas de circuit de coût strictement négatif

BCC est douteuse. Imaginons par exemple des graphes à n sommets et $\sqrt{n}$ arcs (plus précisément, à k^2 sommets et k arcs, avec k tendant vers $+\infty$). Le même problème se présente alors. En fait, il se présente même si on ignore l'initialisation.

Que se passe-t-il si l'on ne suppose plus que G ne contient pas d'arc de coût strictement négatif ? L'algorithme de Dijkstra, étant un cas particulier de l'algorithme de Ford, reste correct. Mais, comme on l'a dit à la section 8.5, il est de complexité exponentielle au pire, et ce cas le pire arrive effectivement. L'exemple, dû à Johnson (D. B. Johnson, *A Note on Dijkstra's Shortest Path Algorithm*, J. ACM 20(3) :385-388, 1973) est présenté en figure 6. Le sommet `s` est complètement à gauche, et les coûts des arcs sont affichés à côté des arcs concernés. Les arcs $2i+1 \to 2i+2$ ont un coût -2^{k-i}, les arcs $2i \to 2i+2$ ont un coût égal à -2^{k-i-1}, et les arcs $2i \to 2i+1$ ont un coût nul.

Le graphe de Johnson ne vérifie pas la contrainte de n'avoir aucun arc de coût strictement négatif. En fait, *tous* ses arcs sont de coût strictement négatif, ou nul. En revanche, il n'a aucun *circuit* de coût strictement négatif, n'ayant pas de circuit du tout.

L'arbre des chemins de coûts minimums n'a qu'un branche, représentée en bleu, et qui passe par les sommets `0`, `1`, `2`, ..., `2k`. Le chemin le plus court (de longueur, et non de coût, le plus faible), est le chemin `0`, `2`, `4`, ..., `2k` du bas. L'algorithme de Dijkstra va systématiquement choisir d'explorer de préférence les arcs du bas en partant des sommets pairs, et nécessitera au moins $2^k - 1$ étapes de calcul.

8.7 Le cas des graphes sans circuit

On a un algorithme encore plus efficace si G est sans circuit (de coût négatif ou non). Dans ce cas, il suffit d'effectuer un tri topologique (inverse) de G, ou plutôt du sous-graphe induit par Reach(s). Supposons en effet les sommets de Reach(s) triés topologiquement, sous forme d'une énumération $x_1, x_2, \cdots, x_k$; on a en particulier $x_k = s$. Par définition, si $x_j \to x_i$ alors $j \geq i$, donc $j > i$ puisque G n'a pas de circuit, donc pas de boucle. Dans ce cas, on a la modification suivante des équations de Bellman du lemme 8.4.

Lemme 8.12 *Si $x_1, x_2, \cdots, x_k$ est un tri topologique de G, et si $s = x_k$, alors pour tout $i \in \{1, \cdots, k\}$,*

$$d(s, x_i) = \inf\{d(s, x_j) + c(x_j, x_i) \mid j > i, x_j \to x_i\}.$$

Démonstration. On a $d(s, x_i) = \inf\{d(s, x_j) + c(x_j, x_i) \mid x_j \to x_i\}$. Mais si $x_j \to x_i$ alors $j > i$. □

La formule du lemme 8.12, aussi appelée équation de Bellman dans la littérature, permet de calculer $d(s, x_i)$ par récurrence décroissante sur i. L'algorithme ainsi obtenu est appelé *algorithme de Bellman* dans le BBC. Il ne faut pas le confondre avec l'algorithme de Bellman-Ford.

algorithme de Bellman

Sa complexité est la somme de :

- la complexité du calcul d'un tri topologique inverse de Reach(s), $O(m + n)$; on utilise pour cela l'algorithme `dfs_2` de la table 6 (avec une petite modification, voir plus bas), appelé sur le sommet s ;
- le parcours des sommets obtenus par ce tri topologique inverse $(x_1, x_2, \cdots, x_k)$ par ordre d'indices décroissants, calculant $d(s, x_i)$ par la formule du lemme 8.12, en temps $O(\sum_{i=1}^{k}(\#\texttt{succ}(x_i) + 1)) = O(n + m)$.

Pour effectuer ce dernier parcours, nous avons besoin de disposer de la liste $x_1, x_2, \cdots, x_k$ dans un tableau `sommets_tries`. Or l'algorithme `dfs_2` ne nous donne le tri topologique qu'implicitement, sous forme d'une fonction de numérotation `rto`. On pourrait penser appeler une procédure de tri, en temps $O(n \log n)$ pour obtenir `sommets_tries`, mais il y a moins coûteux : il suffit de modifier l'algorithme `dfs_2` et d'y ajouter l'affectation suivante en dernière ligne, juste après l'affectation à `rto(u)` :

```
sommets_tries [rto(u)] := u;
```

Théorème 8.13 *L'algorithme de Bellman décrit ci-dessus calcule correctement les distances minimums d'un sommet s de G à tout sommet, à condition que G n'ait aucun circuit. Sa complexité est en $O(m + n)$.*

Comme toujours, ce résultat suppose que la complexité des opérations de base sur les réels sont en temps constant ; voir la remarque 7.26.

On résume les différents algorithmes de calculs de distances minimums à sommet de départ s fixé, en fonction des conditions sous lesquels ils sont corrects, et de leurs complexités, dans la table 17.

8.8 Retour sur les algorithmes de distances minimums entre tous couples de sommets : l'algorithme de Johnson

L'algorithme de Johnson (Donald B. Johnson, *Efficient Algorithms for Shortest Paths in Sparse Networks*, J. ACM 24(1) : 1–13, 1973) combine les algorithmes de Bellman-Ford et de Dijkstra, avec une technique de réétiquetage d'arcs, pour obtenir un algorithme meilleur que l'algorithme de Floyd.

	détecte les circuits <0?	cas général	pas de circuit <0	pas d'arc<0	pas de circuit
Bellman-Ford	oui	$\Theta(n(m+n))$	$\Theta(n(m+n))$	$\Theta(n(m+n))$	$\Theta(n(m+n))$
Ford (FIFO)	non	–	$\Theta(n(m+n))$	$\Theta(n(m+n))$	$\Theta(n(m+n))$
Ford (LIFO)	non	–	expo.	expo.	expo.
Dijkstra	non	–	expo.	$\Theta(m+n \log n)$	expo.
Bellman	non	–	–	–	$\Theta(m+n)$

TABLE 17 – Une vue synthétique des algorithmes de calculs de distances minimums à sommet de départ fixé (on suppose les opérations arithmétiques de base en temps constant)

Soit $G \stackrel{\text{def}}{=} (S, A, c\colon A \to \mathbb{R})$ un graphe valué. On fabrique un nouveau graphe valué $G_* \stackrel{\text{def}}{=} (S_*, A_*, c_*)$ où :

— S_* est l'union disjointe de S et d'un nouveau sommet $*$;

— $A_* \stackrel{\text{def}}{=} A \cup \{(*, u) \mid u \in S\}$, autrement dit les arcs de G_* sont ceux de G, plus un arc de $*$ à chacun des sommets de G ;

— $c_*(e) \stackrel{\text{def}}{=} c(e)$ pour tout arc $e \in A$, et $c_*(*, u) \stackrel{\text{def}}{=} 0$ pour tout $u \in S$. (Note : j'écris $c_*(u, v)$ plutôt que l'inélégant $c_*((u, v))$.)

C'est la construction familière du graphe G_*, avec la définition d'une fonction de coût c_* en plus.

On note que si G n'a pas de circuit de coût strictement négatif, alors G_* non plus. En effet, il n'y a aucun arc entrant sur $*$, donc tous les circuits de G_* sont en fait des circuits de G.

Supposons que G, et donc G_*, n'a pas de circuit de coût strictement négatif. On note $h(u)$, pour tout $u \in S_*$, le coût minimum d'un chemin de $*$ à u dans G_*. On note que ceci est bien défini.

On définit un nouveau graphe $G_{**} \stackrel{\text{def}}{=} (S_*, A_*, c_{**})$, de mêmes sommets et mêmes arcs que G_*, mais avec $c_{**}(u, v) \stackrel{\text{def}}{=} c_*(u, v) + h(u) - h(v)$.

Lemme 8.14 *Si G n'a pas de circuit de coût strictement négatif, alors G_{**} n'a pas d'*arc *de coût strictement négatif.*

Démonstration. Soit $u \to v$ un arc quelconque de G_{**} (ou de G_*, c'est pareil). Il existe un chemin $* \to^* u$ dans G_* de coût $h(u)$. Le chemin $* \to^* u \to v$ obtenu en concaténant l'arc $u \to v$ a un coût $h(u) + c_*(u, v)$, qui est $\geq h(v)$, par minimalité de $h(v)$. Mais cette inégalité signifie que $c_{**}(u, v) \geq 0$. □

L'algorithme de Johnson, dans sa forme la plus simple, calcule h grâce à l'algorithme de Bellman-Ford, puis G_{**}, et calcule ensuite toutes les distances minimum

entre sommets de G_{**} en utilisant $O(n)$ fois l'algorithme de Dijkstra, autant de fois qu'il y a de sommets de départ `s` possibles. Les distances minimum dans le graphe G de départ s'en déduisent grâce à la proposition suivante.

Proposition 8.15 *Soit d la fonction qui à tout couple (u, v) de sommets de G associe le coût minimum d'un chemin de u à v dans G. Soit D la fonction qui à tout couple (u, v) de sommets de G_{**} associe le coût minimum d'un chemin de u à v dans G_{**}. Pour tous $u, v \in S$, $d(u, v) = D(u, v) + h(v) - h(u)$.*

Démonstration. Tous les chemins de u à v dans G sont aussi des chemins dans G_{**}, et tous les chemins de u à v dans G_{**} sont aussi dans G, car u et v sont dans S, et qu'il n'y a pas d'arc entrant sur $*$ dans G_{**}. En particulier, s'il n'y a pas de chemin de u à v (dans G, et dans G_{**}), alors $D(u, v)+h(v)-h(u) = +\infty = d(u, v)$. (On note que $h(v)$ et $h(u)$ ne sont pas infinis, en fait ce sont des réels négatifs ou nuls.) On suppose dorénavant qu'il existe un chemin de u à v dans G, et donc dans G_{**}.

Si $u = u_0 \to u_1 \to \cdots \to u_k = v$ est un chemin γ de u à v dans G, son coût (dans G) est $\sum_{i=1}^{k} c(u_{i-1}, u_i)$, et son coût dans G_{**} est $\sum_{i=1}^{k} c_{**}(u_{i-1}, u_i) = \sum_{i=1}^{k} c(u_{i-1}, u_i) + h(u) - h(v)$ (en annulant les termes $h(u_i)$ deux à deux).

Donc si γ est de coût minimal $d(u, v)$ dans G, il est de coût $d(u, v)+h(u)-h(v)$ dans G_{**}. Par minimalité de D, $D(u, v) \leq d(u, v) + h(u) - h(v)$, donc $d(u, v) \geq D(u, v) + h(v) - h(u)$.

Réciproquement, si γ est de coût minimal $D(u, v)$ dans G_{**}, notons que comme u et v sont dans S et qu'il n'y a pas d'arc entrant sur $*$ dans G_{**}, tous les sommets u_i sont en fait dans S. Alors γ est de coût $D(u, v) + h(v) - h(u)$ dans G, ce qui montre l'inégalité inverse $d(u, v) \leq D(u, v) + h(v) - h(u)$. □

L'algorithme de Johnson qui en résulte est montré à la table 18. La fonction `fresh` appelée à la ligne 1 est supposée retourner un sommet frais, c'est-à-dire hors de `S`. On suppose aussi que l'ensemble `S` des sommets de G est représenté sous forme de liste. À la ligne 2, `S*` vaut donc l'ensemble $S \cup \{*\}$, représenté sous forme de liste. Représenter l'ensemble des sommets sous forme de liste est classique, mais nous permet surtout de représenter la liste des successeurs de `*` dans G_* à la ligne 3 comme la liste `S` elle-même. Le graphe G_* est défini à la ligne 5. On notera que `succ*` et `c*` sont codées comme des fonctions : nous avons, de fait, toujours codé les fonctions `succ` et c comme des fonctions dans ces notes, et ceci nous évite ici la tâche malaisée d'avoir à les convertir en tableaux, si nous les avions codées comme des tableaux, comme il est usuel de le faire. On implémente la correction de distances, passant de D à d comme indiqué à la proposition 8.15, à la ligne 11.

Théorème 8.16 *L'algorithme de Johnson de la table 18 calcule correctement en `d` la fonction des distances minimums entre tout couple de sommets d'un graphe G à n sommets donné en entrée, si G n'a aucun circuit de coût strictement négatif. Sa complexité en temps est de $O(nm + n^2 \log n)$.*

```
   fun johnson (G as (S, succ, c), n) =
1      let * = fresh() in
2      let S* = *::S in
3      let fun succ* (x,y) = if x=* then S else succ (x) in
4      let fun c* (u,v) = if u=* then 0 else c (u,v)) in
5      let G* = (S ∪ {*}, succ*, c*) in
6      let h = bellman-2 (G*, *, n+1) in (* algorithme de Bellman-Ford *)
7      let fun c** (u,v) = c* (u,v)+h[u]-h[v] in
8      let G** = (S*, succ*, c**) in
9      for each s ∈ S do
10              D [s] = ford (G**, s, n+1) ; (* algorithme de Dijkstra
                                                = Ford avec tas de Fibonacci *)
11     let fun d (u, v) = D[u][v]+h[v]-h[u] in
12         return d;
```

TABLE 18 – L'algorithme de Johnson pour les distances minimum entre tous couples de sommets d'un graphe sans circuit de poids strictement négatif

Démonstration. La correction vient du lemme 8.14 et de la proposition 8.15. La complexité est celle de Bellman-Ford, soit $O(n(m+n))$ (voir le théorème 8.6, sachant que `bellman-1` et `bellman-2` ont exactement la même complexité en temps), plus n fois le temps pris par l'algorithme de Dijkstra à la ligne 10 (majoré par une constante fois $m + n\log n$, par le théorème 8.11) ; n fois, car autant de fois qu'on parcourt la boucle sur `s` en lignes 9-10. Le temps total est donc en $O(n(m+n)) + O(nm + n^2\log n) = O(nm + n^2\log n)$. □

Remarque 8.17 *On peut modifier l'algorithme de Johnson pour qu'il détecte les circuits de poids négatifs. Il suffit pour cela d'utiliser* `bellman-1-detect` *(table 15) à la place de* `bellman-2` *à la ligne 6 de la table 18.*

Remarque 8.18 *Tous les algorithmes de la section 8 ont des complexités qui sont données en supposant les opérations de base sur les réels en temps constant, et ceci vaut donc aussi pour l'algorithme de Johnson. Voir la remarque 7.26 pour les précautions d'usage.*

9 Réseaux, flots et coupes

Définition 9.1 *Un* réseau *est le donné d'un graphe valué* $G \stackrel{def}{=} (S, A, c_{\max})$ *sans boucle, où* $c_{\max}$ *est à valeurs dans* $\mathbb{R}_+$*, et de deux sommets* s *(la* source*) et* p *(le* puits*) tels que :* réseau source puits

— $s \neq p$;

— *tout sommet u est accessible depuis s (il y a un chemin $s \to^* u$),*

— *et est coaccessible depuis p (il y a un chemin $u \to^* p$).*

Pour tout arc e, on appelle $c_{\max}(e)$ la capacité maximale *de e.* capacité maximale

Le BBC (section 8.2) considère un modèle plus complexe, et plus complet : il y a aussi une fonction $a\colon A \to \mathbb{R}_+$ qui à chaque arc e associe une capacité *minimale*, il y a aussi des coûts associés à chaque arc. La définition ci-dessus, plus simple, nous suffira dans la mesure où nous ne nous intéresserons qu'à des problèmes de flots maximaux, et pas par exemple de flots maximaux de coût minimum, par exemple.

La définition du BBC est aussi légèrement différente. D'abord, $c_{\max}$ y est à valeurs rationnelles (l'explication en est donné à la remarque en bas de la page 261 du BBC). Ensuite, la définition du BBC n'inclut ni s ni p. On retrouve la définition du BCC à partir de la nôtre en ajoutant un arc de p à s, de capacité maximale suffisamment grande. La définition avec s et p est plus courante dans la littérature.

9.1 Flots et coupes

Définition 9.2 *Un* flot *dans un réseau (G, s, p), où $G \stackrel{def}{=} (S, A, c_{\max})$, est une fonction $f\colon A \to \mathbb{R}_+$ telle que :* flot

1. *Pour tout sommet u de G différent de s et de p, $\sum_{v/v\to u} f(v \to u) = \sum_{w/u\to w} f(u \to w)$;*
2. *pout tout arc e, $f(e) \leq c_{\max}(e)$.*

La condition du point 1 est la condition de *conservation du flot en u*. La condition du point 2 est la condition d'*admissibilité*. conservation du flot en u admissibilité

Les notations utilisées au point 1 sont, je l'espère, claires. Le point 1 exprime que le *flot entrant* en u, qui est la somme des flots sur les arcs d'extrémité u, égale le *flot sortant* de u, c'est-à-dire la somme des flots sur les arcs d'origine u. On ne le demande par pour la source s ni pour le puits p : l'idée est que s engendre un flot sortant, et que p absorbe un flot entrant. flot entrant flot sortant

Remarque 9.3 *Rien dans la définition ne demande à ce qu'il n'y ait pas d'arc $u \to s$ ou $p \to v$. En fait, il est en général possible qu'il y ait non seulement un flot sortant mais aussi* entrant *sur s, et de même qu'il y ait un flot entrant mais aussi* sortant *sur p.*

Définition 9.4 *Une* coupe *d'un réseau (G, s, p) est un ensemble de sommets E* coupe

de G contenant s mais pas p. Pour tout fonction $f\colon A \to \mathbb{R}_+$, où A est l'ensemble des arcs de G, et notamment pour tout flot, on pose :

$$\mathrm{out}(f, E) \stackrel{def}{=} \sum_{(u,w)\in A, u\in E, w\notin E} f(u, w)$$
$$\mathrm{in}(f, E) \stackrel{def}{=} \sum_{(v,u)\in A, v\notin E, u\in E} f(v, u).$$

Les arcs (u, w) tels que $u \in E$ et $w \notin E$ sont les arcs sortants de E, *les arcs (v, u) tels que $v \notin E$ et $u \in E$ sont les* arcs entrants de E.

arcs sortants de E

arcs entrants de E

Lemme 9.5 *Pour tout flot f sur un réseau (G, s, p), la quantité $\mathrm{out}(f, E) - \mathrm{in}(f, E)$ est la même pour toutes les coupes E.*

Démonstration. Par définition d'un flot, pour tout $u \in E \setminus \{s\}$ (donc u est différent de p, aussi, car une coupe E ne contient pas p),

$$\sum_{v/v\to u} f(v \to u) = \sum_{w/u\to w} f(u \to w).$$

On somme maintenant sur tous les $u \in E \setminus \{s\}$. Appelons G la somme des côtés gauches, et D la somme des côtés droits ; en particulier, $G = D$, ce que nous utiliserons plus loin.

En utilisant des notations telles que $\sum_{v\notin E\to u\in E}$ pour désigner une somme sur tous les arcs d'origine un sommet hors de E et d'extrémité un somme dans E, et d'autres notations similaires, on a :

$$\begin{aligned} G &= \sum_{v\notin E\to u\in E\setminus\{s\}} f(v \to u) + \sum_{v\in E\to u\in E} f(v \to u) \\ &= \sum_{v\notin E\to u\in E} f(v \to u) + \sum_{v\in E\to u\in E} f(v \to u) - \sum_{v\notin E\to s} f(v \to s) \\ &\qquad \text{puisque } s \text{ fait partie des sommets de } E \\ &= \mathrm{in}(f, E) + \sum_{v\in E\to u\in E} f(v \to u) - \mathrm{in}(f, \{s\}). \end{aligned}$$

De façon similaire,

$$\begin{aligned} D &= \sum_{u\in E\setminus\{s\}\to w\notin E} f(u \to w) + \sum_{u\in E\setminus\{s\}\to w\in E} f(u \to w) \\ &= \sum_{u\in E\to w\notin E} f(u \to w) + \sum_{u\in E\setminus\{s\}\to w\in E} f(u \to w) - \sum_{w\notin E} f(s \to w) - \sum_{w\in E} f(s \to w) \\ &= \mathrm{out}(f, E) + \sum_{u\in E\setminus\{s\}\to w\in E} f(u \to w) - \mathrm{out}(f, \{s\}). \end{aligned}$$

Comme $G = D$, et que la somme $\sum_{v \in E \to u \in E} f(v \to u)$ du milieu de notre dernière expression de G est égale à la somme $\sum_{u \in E \setminus \{s\} \to w \in E} f(u \to w)$ du milieu de notre dernière expression de D, on a :

$$\mathrm{in}(f, E) - \mathrm{in}(f, \{s\}) = \mathrm{out}(f, E) - \mathrm{out}(f, \{s\}),$$

donc :

$$\mathrm{out}(f, E) - \mathrm{in}(f, E) = \mathrm{out}(f, \{s\}) - \mathrm{in}(f, \{s\}),$$

ce qui démontre le lemme. □

Ceci donne un sens à la définition suivante.

Définition 9.6 *La* valeur d'un flot *f sur un réseau (G, s, p) est :* valeur d'un flot

$$\mathrm{val}(f) \stackrel{def}{=} \mathrm{out}(f, E) - \mathrm{in}(f, E),$$

pour n'importe quelle coupe E du réseau G, par exemple $\{s\}$ ou $A \setminus \{p\}$, A étant l'ensemble des arcs de G.

9.2 Flots maximaux, coupes minimales

Définition 9.7 *La* valeur d'une coupe *E d'un réseau (G, s, p), où $G \stackrel{def}{=} (S, A, c_{\max})$,* valeur d'une coupe *est :*

$$\mathrm{b}(E) \stackrel{def}{=} \mathrm{out}(c_{\max}, E) = \sum_{u \in E \to w \notin E} c_{\max}(u \to w),$$

la somme des capacités maximales des arcs sortants de E.

Lemme 9.8 *Pour tout flot f sur un réseau (G, s, p), pour tout coupe E,* $\mathrm{val}(f) \leq \mathrm{b}(E)$.

Démonstration. On a :

$$\begin{aligned} \mathrm{val}(f) &= \mathrm{out}(f, E) - \mathrm{in}(f, E) \\ &\leq \mathrm{out}(f, E) \\ &\leq \mathrm{out}(c_{\max}, E) && \text{car } f \leq c_{\max} \text{ (admissibilité)} \\ &= \mathrm{b}(E). && \square \end{aligned}$$

En particulier, $\max_{f \text{ flot}} \mathrm{val}(f) \leq \min_{E \text{ coupe}} \mathrm{b}(E)$. Le théorème max-flow min-cut, que nous verrons plus bas (théorème 9.32), énonce que les deux côtés de l'inégalité sont en fait égaux. Il est en général attribué à Ford et Fulkerson (L. R. Ford, Jr. et D. R. Fulkerson, *Maximal Flow through a Network*, Can. J. Math. 8(3) :399-404, 1956, version journal d'un rapport de la RAND corp. de novembre

1954), bien qu'il ait été découvert bien avant par Kantorovich (L. V. Kantorovich, *Matematicheskie metody organizatsii i planirovaniia proizvodstva*, Leningrad : State Publishing House, 1939, traduit sous le titre *Mathematical Methods of Organizing and Planning Production*, Management Science 6(4) :363–422, 1960). Ford et Fulkerson ont donné une approche plus algorithmique à la question, et nous déduirons le théorème max-flow min-cut de la correction de leur algorithme.

Dans la suite, un flot f maximisant $\mathrm{val}(f)$ sera appelé un *flot maximal*. Une coupe E minimisant $\mathrm{b}(E)$ sera appelée une *coupe minimale*.

flot maximal

coupe minimale

Remarque 9.9 *Étant donné un flot f sur (G, s, p), il suffit d'exhiber une coupe E telle que $\mathrm{val}(f) = \mathrm{b}(E)$ pour garantir que f est un flot maximal, par le lemme 9.8.*

De façon symétrique, étant donnée une coupe E, il suffit d'exhiber un flot f tel que $\mathrm{val}(f) = \mathrm{b}(E)$ pour garantir que E est une coupe minimale.

9.3 Le graphe d'écart, et les chemins améliorants

L'outil principal que nous utiliserons est le graphe d'écart défini ci-dessous. La dénomination d'un arc passant ou inerte n'est pas standard dans la littérature.

Définition 9.10 *Soit (G, s, p) un réseau et f un flot sur ce réseau, ou plus généralement une fonction des arcs de G vers $\mathbb{R}_+$ telle que $0 \leq f(e) \leq c_{\max}(e)$ pour tout arc e de G.*

Un arc $x \to y$ de G est libre *(par rapport à f) si $f(x, y) < c_{\max}(x, y)$, et* saturé *sinon.*

libre

saturé

Un arc $x \to y$ de G est passant *(par rapport à f) si $f(x, y) > 0$, et est* inerte *sinon.*

passant

inerte

Le graphe d'écart *G_f de f est le graphe valué ayant les mêmes sommets que G, et avec deux types d'arcs, que l'on notera $\to_f$ pour rappeler la dépendance en le flot f, et une fonction valeur r_f, définis comme suit :*

graphe d'écart

- *pour tout arc libre $x \to y$ de G, un arc $x \to_f y$, dit* arc conforme*, de valeur $r_f(x \to_f y) \stackrel{def}{=} c_{\max}(x, y) - f(x, y)$;*
- *pour tout arc passant $x \to y$ de G, un arc $y \to_f x$, dit* arc non conforme*, de valeur $r_f(y \to_f x) \stackrel{def}{=} f(x, y)$.*

arc conforme

arc non conforme

Dans le premier cas, on dit que $x \to_f y$ est le représentant conforme *de l'arc libre $x \to y$. Dans le deuxième cas, $y \to_f x$ est le* représentant non conforme *de l'arc passant $x \to y$.*

représentant conforme

représentant non conforme

On notera que les arcs non conformes $y \to_f x$ sont renversés, comparés à l'arc passant $x \to y$ dont ils sont issus.

Remarque 9.11 *Le graphe d'écart est en fait un multigraphe : il peut très bien y avoir deux arcs $x \to_f y$ pour un même couple de sommets x et y. L'un est alors*

nécessairement conforme et l'autre non conforme. Nous conservons cependant l'appellation traditionnelle de graphe *d'écart. Mais nous devrons noter que la notation $r_f(x \to_f y)$ est ambiguë, dans la mesure où il peut y avoir deux arcs de x à y dans le graphe d'écart G_f. Le contexte nous permettra toujours de déterminer si nous parlons du coût de l'arc conforme ou de l'arc non conforme de x à y.*

Remarque 9.12 *Tout arc de G a 0, 1 ou 2 représentants. Le cas où $x \to y$ n'a aucun représentant est celui d'un arc à la fois saturé et inerte, ce qui ne peut arriver que si $c_{\max}(x \to y) = 0$, une situation qu'on pourrait cherche à éviter en demandant que toutes les capacités maximales soit strictement positives, et en éliminant les arcs de capacité maximale nulle — mais ce n'est pas nécessaire.*

Si $x \to y$ a un représentant, il peut être conforme ou non conforme.

Le seul cas où $x \to y$ a deux représentants est celui où $x \to y$ est à la fois libre et passant, c'est-à-dire où $0 < f(x \to y) < c_{\max}(x \to y)$. Dans ce cas, les deux représentants sont dits conjoints. conjoints

Lemme 9.13 *Tout arc de G_f est représentant d'un arc unique de G. Pour tout arc e de G, la somme des valeurs de r_f sur les 0, 1 ou 2 représentants de e vaut toujours $c_{\max}(e)$.*

Définition 9.14 *Une* chemin améliorant *μ pour un flot f sur un réseau (G, s, p) est un chemin élémentaire de s à p dans le graphe d'écart G_f.* chemin améliorant

Il semble que la condition que le chemin soit *élémentaire* ait été omise dans le BBC. Elle est cependant cruciale. En effet, dans un chemin élémentaire, tout sommet apparaît au plus une fois, et donc il en est de même pour les arcs. De plus, et ceci mérite un lemme :

Lemme 9.15 *On ne peut pas trouver à la fois le représentant conforme et le représentant non conforme d'un même arc de G dans aucun chemin améliorant μ.*

Démonstration. Ces deux représentants auraient leurs sommets en commun, et l'un deux au moins serait donc répété sur le chemin μ. □

Définition 9.16 *L'*amélioration *$\alpha(\mu)$ d'un chemin améliorant μ est le plus petit coût (tel que calculé par r_f) d'un arc de μ.* amélioration

Le lemme suivant montre qu'un chemin améliorant permet de fabriquer un nouveau flot $f.\mu$ à partir d'un flot f, de valeur supérieure (ou égale).

Lemme 9.17 *Soit μ un chemin améliorant pour un flot f sur un réseau (G, s, p). La fonction $f.\mu$ qui à tout arc e de G associe :*

— *$f(e)$ si aucun représentant de e n'apparaît dans μ,*

— $f(e) + \alpha(\mu)$ *si* e *a un représentant conforme* e'*, et* e' *est l'un des arcs de* μ*,*

— $f(e) - \alpha(\mu)$ *si* e *a un représentant non conforme* e''*, et* e'' *est l'un des arcs de* μ*,*

est un flot, et $\mathrm{val}(f.\mu) = \mathrm{val}(f) + \alpha(\mu)$.

Démonstration. Soit u un sommet différent de s et de p. Si u n'est pas un sommet de μ, alors $f.\mu$ et f ont les mêmes valeurs sur tous les arcs d'origine ou d'extrémité u, donc l'équation de conservation du flot de $f.\mu$ en u est évidente. Explicitement,

$$\begin{aligned}\sum_{v/v\to u}(f.\mu)(v\to u) &= \sum_{v/v\to u} f(v\to u)\\ &= \sum_{w/u\to w} f(u\to w)\\ &= \sum_{w/u\to w}(f.\mu)(u\to w).\end{aligned}$$

Supposons donc que μ passe par le sommet u, exactement une fois puisque μ est un chemin élémentaire. Comme $u \neq s, p$ et qu'un chemin améliorant est d'origine s et d'extrémité p, μ est de la forme $s \to_f^* v_0 \to_f u \to_f w_0 \to_f^* p$. Soit a l'arc de G dont $v_0 \to_f u$ est un représentant, et b celui dont $u \to_f w_0$ est un représentant. On cherche à montrer que $\sum_{v/v\to u}(f.\mu)(v\to u) = \sum_{w/u\to w}(f.\mu)(u\to w)$. On a quatre cas, selon que $v_0 \to_f u$ est conforme ou non, et selon que $u \to_f w_0$ est conforme ou non.

— Si $v_0 \to_f u$ et $u \to_f w_0$ sont conformes, alors $a = v_0 \to u$ et $b = u \to w_0$, et l'on a :

$$\begin{aligned}\sum_{v/v\to u}(f.\mu)(v\to u) &= \sum_{v/v\to u, v\neq v_0}(f.\mu)(v\to u) + (f.\mu)(v_0\to u)\\ &= \sum_{v/v\to u, v\neq v_0} f(v\to u) + f(v_0\to u) + \alpha(\mu)\\ \sum_{w/u\to w}(f.\mu)(u\to w) &= \sum_{w/u\to w, w\neq w_0}(f.\mu)(u\to w) + (f.\mu)(u\to w_0)\\ &= \sum_{w/u\to w, w\neq w_0} f(u\to w) + f(u\to w_0) + \alpha(\mu),\end{aligned}$$

qui sont deux quantités égales, car f est un flot.

— Si $v_0 \to_f u$ est conforme et $u \to_f w_0$ est non conforme, alors $a = v_0 \to u$ et $b = w_0 \to u$ (l'arc est renversé), donc a et b interviennent tous les deux

dans la première somme :

$$\begin{aligned}\sum_{v/v\to u}(f.\mu)(v\to u) &= \sum_{v/v\to u, v\neq v_0, w_0}(f.\mu)(v\to u)+(f.\mu)(v_0\to u)+(f.\mu)(w_0\to u)\\ &= \sum_{v/v\to u, v\neq v_0, w_0} f(v\to u)+f(v_0\to u)+\alpha(\mu)+f(w_0\to y)-\alpha(\mu)\\ &= \sum_{v/w\to u} f(v\to u),\end{aligned}$$

ce qui est égal à $\sum_{w/u\to w} f(u\to w)$, puisque f est un flot, et donc à $\sum_{w/u\to w}(f.\mu)(u\to w)$ puisque ni a ni b n'est un arc d'origine u (μ est un chemin élémentaire, et de toute façon un réseau n'a pas de boucle). Notons que comme $u\to_f w_0$ est non conforme, on a $(f.\mu)(w_0\to u) = f(w_0\to u)-\alpha(\mu)$, avec un signe moins.

— Si $v_0\to_f u$ est non conforme et $u\to_f w_0$ est conforme, alors c'est la second somme que l'on doit considérer :

$$\begin{aligned}\sum_{w/u\to w}(f.\mu)(u\to w) &= \sum_{w/u\to w, w\neq v_0, w\neq w_0}(f.\mu)(u\to w)+(f.\mu)(u\to v_0)+(f.\mu)(u\to w_0)\\ &= \sum_{w/u\to w, w\neq v_0, w\neq w_0} f(u\to w)+f(u\to v_0)-\alpha(\mu)+f(u\to w_0)+\alpha(\mu)\\ &= \sum_{w/u\to w} f(u\to w) = \sum_{v/v\to u} f(v\to u) = \sum_{v/v\to u}(f.\mu)(v\to u).\end{aligned}$$

— Si $v_0\to_f u$ et $u\to_f w_0$ sont tous les deux non conformes, alors $a = u\to v_0$ et $b = w_0\to u$, donc :

$$\begin{aligned}\sum_{v/v\to u}(f.\mu)(v\to u) &= \sum_{v/v\to u, v\neq w_0}(f.\mu)(v\to u)+(f.\mu)(w_0\to u)\\ &= \sum_{v/v\to u, v\neq w_0} f(v\to u)+f(w_0\to u)-\alpha(\mu)\\ \sum_{w/u\to w}(f.\mu)(u\to w) &= \sum_{w/u\to w, w\neq v_0}(f.\mu)(u\to w)+(f.\mu)(u\to v_0)\\ &= \sum_{w/u\to w, w\neq v_0} f(u\to w)+f(u\to v_0)-\alpha(\mu),\end{aligned}$$

qui sont donc des quantités égales.

Donc $f.\mu$ vérifie l'équation de conservation du flot.

Vérifions maintenant la condition d'admissibilité. Comme $f.\mu$ et f coincident sur tous les arcs n'ayant aucun représentant appartenant à μ, il suffit de montrer que $0\leq(f.\mu)(e)\leq c_{\max}(e)$ pour tout arc ayant un représentant sur le chemin μ. On rappelle que ce représentant est alors unique.

— Si le représentant e' de e est conforme, $(f.\mu)(e) = f(e) + \alpha(\mu)$; notamment, comme $\alpha(\mu) \geq 0$, on a $(f.\mu)(e) \geq f(e) \geq 0$. Pour ce qui est de l'autre inégalité, $\alpha(\mu)$ étant défini comme un plus petit coût est inférieur ou égal à $r_f(e')$, et $r_f(e') = c_{\max}(e) - f(e)$. Donc $(f.\mu)(e) \leq f(e) + c_{\max}(e) - f(e) = c_{\max}(e)$.

— Si le représentant e'' de e est non conforme, $(f.\mu)(e) = f(e) - \alpha(\mu)$; donc $(f.\mu)(e) \leq f(e) \leq c_{\max}(e)$. On a aussi $\alpha(\mu) \leq r_f(e'') = f(e)$, donc $(f.\mu)(e) \geq f(e) - f(e) \geq 0$.

Il reste à montrer que $\mathrm{val}(f.\mu) = \mathrm{val}(f) + \alpha(\mu)$. On a $\mathrm{val}(f.\mu) = \mathrm{out}(f.\mu, \{s\}) - \mathrm{in}(f.\mu, \{s\}) = \sum_{w/s\to w}(f.\mu)(s \to w) - \sum_{v/v\to s}(f.\mu)(v \to s)$. Comme $\mu \colon s \to_f^* p$ est un chemin élémentaire, il y a exactement un arc $s \to w$ ou $v \to s$ ayant un représentant dans μ, et ce re représentant est le premier arc de μ. Pour tous les autres arcs e de la somme définissant $\mathrm{val}(f.\mu)$, on a $(f.\mu)(e) = f(e)$.

— Si le premier arc e' de μ est un représentant, nécessairement conforme, d'un arc $s \to w$, alors $(f.\mu)(s \to w) = f(s \to w) + \alpha(\mu)$, donc $\mathrm{val}(f.\mu) = \mathrm{val}(f) + \alpha(\mu)$.

— Si le premier arc s'' de μ est un représentant, nécessairement d'un arc non conforme, d'un arc $v \to s$, alors $(f.\mu)(v \to s) = f(v \to s) - \alpha(\mu)$, donc $\mathrm{val}(f.\mu) = \mathrm{val}(f) - (-\alpha(\mu)) = \mathrm{val}(f) + \alpha(\mu)$. □

Corollaire 9.18 *Si un flot f sur un réseau (G, s, p) a un chemin améliorant μ, alors* $\mathrm{val}(f)$ *n'est pas maximum.*

Démonstration. Par le lemme 9.17, $\mathrm{val}(f.\mu) = \mathrm{val}(f) + \alpha(\mu)$. Or $\alpha(\mu)$ est un minimum de valeurs de r_f, qui sont toutes strictes positives par définition. □

L'implication du corollaire 9.18 est en fait une équivalence, la direction réciproque étant donnée par le lemme suivant.

Lemme 9.19 *S'il n'existe pas de chemin améliorant pour un flot f sur un réseau (G, s, p), alors l'ensemble E des sommets accessibles depuis s dans le graphe d'écart G_f est une coupe et* $\mathrm{val}(f) = \mathrm{b}(E)$.

Démonstration. Le sommet p ne peut pas être dans E, sinon il y aurait un chemin $s \to_f^* p$ dans G_f, donc un chemin élémentaire (lemme 4.3, de Kőnig), c'est-à-dire un chemin améliorant. Donc E est une coupe.

Par définition de E, on a : $(*)$ il n'y a pas d'arc $u \in E \to_f v \notin E$.

Considérons un arc $e \colon v \notin E \to u \in E$ dans G. Si $f(e) \neq 0$, alors il y aura un arc non conforme $u \to_f v$ dans G_f, ce qui n'est pas possible par $(*)$. Donc $\mathrm{in}(f, E) = \sum_{v\notin E\to u\in E} f(e) = 0$.

Considérons maintenant un arc $e \colon u \in E \to w \notin E$ dans G. Si $f(e) \neq c_{\max}(e)$, alors il y aura un arc conforme $u \to_f w$ dans G_f, ce qui n'est pas possible par $(*)$. Donc $\mathrm{out}(f, E) = \sum_{u\in E\to w\notin E} f(e) = \sum_{u\in E\to w\notin E} c_{\max}(e) = \mathrm{out}(c_{\max}, E) = \mathrm{b}(E)$.

On a donc $\mathrm{val}(f) = \mathrm{out}(f, E) - \mathrm{in}(f, E) = \mathrm{b}(E)$. □

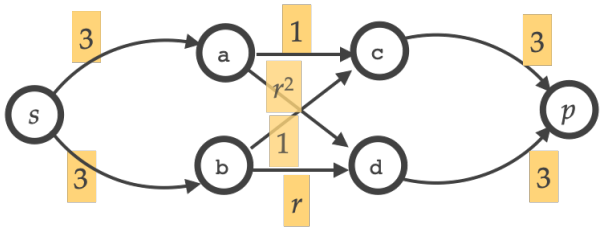


FIGURE 7 – Le réseau de Zwick ; $\lambda \approx 0,216757$ est l'unique racine réelle de $1-5x+2x^2-x^3$, $r=(1+\sqrt{1-4\lambda})/2 \approx 0,682378$; les capacités maximum sont affichées sur fond orange

9.4 L'algorithme de Ford-Fulkerson, et la question de sa terminaison

L'algorithme de Ford-Fulkerson, de façon abstraite, est simple : on initialise le flot f au flot nul, puis, tant qu'il existe un chemin améliorant μ dans le graphe d'écart G_f, on met à jour le flot, en remplaçant f par $f.\mu$. Ceci laisse un certain nombre de détails de côté, comme : comment construit-on efficacement un chemin améliorant ? comment choisir les chemins améliorants de sorte à minimiser le nombre de mises à jour du flot ? comment met-on à jour efficacement le flot ? Mais nous avons un problème plus grave à régler d'abord, et c'est celui de la terminaison de l'algorithme.

Ford et Fulkerson avaient déjà observé que leur algorithme pouvait ne pas terminer si les capacités maximum n'étaient pas toutes rationnelles. De façon plus précise, il existe des réseaux, et des stratégies de sélection de chemins améliorants qui font que l'algorithme de Ford-Fulkerson ne termine pas. L'exemple le plus petit est dû à Zwick (U. Zwick, *The Smallest Networks on which the Ford-Fulkerson Maximum Flow Procedure May Fail to Terminate*, Theor. Comp. Sci. 148 :165–170, 1995). Il s'agit du réseau montré à la figure 7. La stratégie consistant à choisir les chemins améliorants s`acbd`p, s`adbc`p, s`bdac`p, s`bcad`p dans cet ordre, et à répéter ces quatre choix à l'infini, ne termine pas.

Il existe cependant une autre stratégie de choix des chemins améliorants permettant de trouver le flot maximum en quatre étapes de mise à jour du flot seulement. Sa valeur est $2+r+r^2 \approx 3,147899$. La coupe minimum est $\{s, \mathtt{a}, \mathtt{b}\}$.

On pourrait imaginer que la suite des flots trouvés dans l'algorithme de Ford-Fulkerson convergerait vers le flot maximum, mais ceci aussi est faux, comme le note Zwick.

9.5 Distances estimées et le graphe d'admissibilité

La première stratégie de sélection des chemins améliorants fournissant un comportement en temps polynomial à l'algorithme de Ford-Fulkerson, et montrant notamment qu'il peut être rendu terminant, est usuellement attribuée à Edmonds

et Karp (J. Edmonds, R. M. Karp, *Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems*, J. ACM 19(2) :248-264, 1972). Mais Dinic (ou Dinitz) avait déjà trouvé cette solution deux ans auparavant, et avec une amélioration algorithmique supplémentaire, que nous présentons sous la forme plus moderne, et de fait plus simple à comprendre et à implémenter, des distances estimées (E. A. Dinic, *Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimation*, Dokl. Akad. Nauk USSR 194(4), 1970, traduit en anglais dans Soviet Math. Doklady 11(5) :1277-1280, 1970). Pour être précis, Dinic utilisait certaines partitions de l'ensemble des sommets ; son algorithme a été popularisé par Even et Itai, et c'est Goldberg, plus tard, qui a remplacé ces partitions par des distances estimées Δ ; les partitions de Dinic se retrouvent comme $\{\Delta^{-1}(\{k\}) \mid 0 \leq k \leq n\}$.

L'idée de Karp et Edmonds est d'explorer l'espace des chemins améliorants par longueurs croissantes. On rappelle que la longueur est le nombre d'arcs. Dinic a montré comme gérer efficacement le recalcul des chemins améliorants après chaque étape d'amélioration du flot grâce au maintien d'une table de distances estimées. Il s'agit d'approximations des distances (longueurs minimales d'un chemin) de tous les sommets au puits p dans le graphe d'écart G_f, qui de plus sera efficace à mettre à jour lorsqu'on mettra à jour f.

Définition 9.20 *Soit f un flot sur un réseau (G, s, p), ou plus généralement une fonction des arcs de G vers $\mathbb{R}_+$ telle que $0 \leq f(e) \leq c_{\max}(e)$ pour tout arc e de G. Une* distance estimée *relative au flot f (ou* sur *le graphe d'écart G_f) est une fonction Δ des sommets de G vers $\mathbb{N}$ telle que :*

distance estimée

1. *$\Delta(p) = 0$;*
2. *pour tout arc $x \to_f y$ du graphe d'écart G_f, $\Delta(x) \leq \Delta(y) + 1$.*

Lemme 9.21 *Pour tout flot f sur un réseau (G, s, p), pour tout distance estimée Δ relative au flot f, pour tout sommet x de G, $\Delta(x)$ est inférieur ou égal à la longueur du plus petit chemin de x à p dans le graphe d'écart G_f (si un tel chemin existe).*

Démonstration. Il suffit de montrer que $\Delta(x) \leq k$ pour n'importe quel chemin $x = x_0 \to_f x_1 \to_f \cdots \to_f x_k = p$ de x à p dans G_f. Par la condition 2 de la définition 9.20, $\Delta(x) = \Delta(x_0) \leq \Delta(x_1) + 1 \leq \Delta(x_2) + 2 \leq \cdots \leq \Delta(x_k) + k = \Delta(p) + k$, et l'on conclut car par la condition 1, $\Delta(p) = 0$. □

On en déduit le corollaire suivant, qui nous servira de critère de terminaison.

Corollaire 9.22 *Pour tout flot f sur un réseau (G, s, p), si Δ est une distance estimée relative au flot f telle que $\Delta(s) \geq n$, alors il n'existe pas de chemin de s à p dans le graphe d'écart G_f, et donc f est un flot de valeur maximum, égale à la valeur de la coupe E formée des sommets accessibles dans le graphe d'écart G_f.*

Démonstration. En utilisant le lemme de Kőnig, le plus petit chemin de s à p, étant élémentaire, est de longueur au plus $n-1$ (au plus n sommets, donc au plus $n-1$ arcs). Si $\Delta(s) \geq n$, par le lemme 9.21, c'est qu'il ne peut pas exister de chemin de s à p dans G_f. Il n'existe donc aucun chemin améliorant, donc f est de valeur maximum et $\mathrm{val}(f) = \mathrm{b}(E)$ par le lemme 9.19. □

L'algorithme de Dinic-Edmonds-Karp va chercher uniquement des chemins améliorants particuliers, appelés chemins admissibles, et définis comme suit.

Définition 9.23 *Soit f un flot sur un réseau (G, s, p), et Δ une distance estimée relative à f. Un arc $x \to_f y$ du graphe d'écart G_f est un* arc admissible *si et seulement $\Delta(x) = \Delta(y) + 1$.*

arc admissible

Un chemin admissible *est un chemin améliorant dont tous les arcs sont admissibles.*

chemin admissible

Remarque 9.24

1. *La longueur d'un chemin admissible vaut exactement $\Delta(s)$. Par le lemme 9.21, sa longueur est inférieure ou égale à celle de tout chemin de s à p dans le graphe d'écart G_f, c'est donc un chemin améliorant* de longueur minimum.
2. *Pour tout sommet x d'un chemin admissible $s = x_0 \to x_1 \to \ldots \to x_k$, on a $\Delta(x) \leq \Delta(s)$, car si $x = x_i$, alors $\Delta(s) = \Delta(x_0) = \Delta(x_1) + 1 = \cdots = \Delta(x_i) + i$. En particulier, si $\Delta(s) < n$, alors $\Delta(x) < n$ pour tout sommet x du chemin admissible.*
3. *Un chemin admissible est la même chose qu'un chemin $\mu\colon s \to_f^* p$ dans le graphe d'écart dont tous les arcs sont admissibles, et on n'a pas besoin d'imposer qu'il soit élémentaire. Il l'est, car si μ est de la forme $s = x_0 \to x_1 \to \ldots \to x_k$, tous les sommets x_i ont des valeurs $\Delta(x_i)$ égales à $\Delta(s) - i$, donc deux à deux distinctes, et par conséquent les sommets x_i sont eux aussi distincts deux à deux.*

Définition 9.25 *Soit f un flot sur un réseau (G, s, p), ou plus généralement une fonction des arcs de G vers $\mathbb{R}_+$ telle que $0 \leq f(e) \leq c_{\max}(e)$ pour tout arc e de G, et Δ une distance estimée relative à f. Le* graphe d'admissibilité *$G_{f,\Delta}$ est le sous-graphe couvrant de G_f dont les arcs sont les arcs admissibles.*

graphe d'admissibilité

9.6 L'algorithme de Dinic et Edmonds-Karp, et le théorème max-flow min-cut

L'algorithme de Dinic-Edmonds-Karp se décrit maintenant, de façon encore relativement abstraite, comme suit.

1. Initialisation : on initialise f au flot nul, et pour tout sommet x de G, on initialise $\Delta(x)$ à la longueur du plus court chemin $x \to_f^* p$ dans le graphe d'écart G_f.

2. Tant que $\Delta(s) < n$, on alterne entre deux opérations :
   (a) trouver un chemin admissible μ, et *améliorer le flot*, c'est-à-dire poser $f := f.\mu$;
   (b) ou *mettre à jour* la distance estimée Δ de sorte à augmenter les distances estimées, sans changer f.

Par le corollaire 9.22, si cet algorithme termine, alors il terminera alors que f sera un flot de valeur maximum. Nous raffinerons plus tard cette procédure en définissant une opération supplémentaire 2(c) permettant de découvrir les chemins admissibles.

L'étape d'initialisation est un calcul de plus court chemin de p à tout sommet dans le graphe d'écart *renversé*. Il suffit pour cela d'effectuer un parcours en largeur, et il se trouve que c'est déjà ce que fait l'algorithme de Ford à files (voir la section 8.4), sur le graphe d'écart renversé donc, et avec des coûts d'arcs tous égaux à 1. Dans ce cas particulier où tous les coûts d'arcs sont égaux à 1, on peut renforcer l'invariant **(Inv 9)**, et réaliser que l'entier k qui y est mentionné est égal à la longueur du plus court chemin de la source (nommée `s` en section 8.4, et égale à p dans notre cas) à tout sommet de w', et que $k-1$ est la longueur du plus court chemin de la source à tout sommet de w, donc que `d`$[x]$ est déjà égale à la longueur du plus court chemin depuis la source vers x, pour tout sommet x de la file `work`. Comme dans l'algorithme de Dijkstra, il s'ensuit que tout sommet `x` une fois extrait aura déjà reçu sa valeur optimale dans `d`[`x`], et ne sera donc plus jamais rajouté à la file `work`. En conséquence, l'algorithme de Ford à files, sur un graphe de coûts d'arcs tous égaux à 1, termine en temps $O(m+n)$.

En pratique, on ne construit pas le graphe d'écart, et on se contente de simuler un parcours sur le graphe d'écart via un algorithme travaillant directement sur le graphe G. Pour ceci, on suppose que G est représenté par listes de successeurs *et de prédécesseurs*. (Sinon, on précalcule les listes de prédécesseurs une fois pour toutes au début, en temps $O(m+n)$, et ceci est laissé en exercice.) Notons `pred`(x) la liste de prédécesseurs d'un sommet x. L'algorithme d'initialisation résultant est montré à la figure 19. On a supprimé les lignes calculant la fonction `p` codant l'arbres des sommets de distances minimum, la source a été renommée `p` (plutôt que `s`), la définition de `improve` a été simplifiée en `Delta`[`x`] + `1` (le coût d'un arc valant 1), mais surtout, `init-delta` prend comme arguments supplémentaires un flot `f`, un fonction $c_{\max}$ dans la variable `cmax`, et on dédouble le parcours des successeurs de `x` (les lignes 9–14 de la table 16) en un parcours des successeurs de `x` via un arc *conforme* de G_f (lignes 9–15 de la table 19) plus un parcours des successeurs de `x` via un arc *non conforme* de G_f (lignes 16–22).

Les lignes 11-12 et 18–19 ont conservé un style proche de celui utilisé pour des lignes similaires dans l'algorithme de Ford. À la ligne 2, on a initialisé les valeurs de `Delta` à `n` plutôt qu'à $+\infty$. Ceci n'entache pas la correction de l'algorithme, la démonstration de la correction de l'algorithme de Ford restant correcte si l'on remplace l'invariant **(Inv 3)** de la section 8.3 par « pour tout sommet x de G

```
   fun init-delta (G as (S, succ, pred), p, f, cmax, n) =
1      let Delta = new(n) in for each x ∈ S do
2                                    Delta[x] := n; (* plutôt que +∞ *)
3                                    (* p[x] := undef : inutile *)
4      Delta[p] := 0;
5      work := empty;
6      push(work,p);
7      while nonempty(work) do
8            let x=pop(work) in
9                for each y ∈ succ(x) do
10                        if f(x,y)<cmax(x,y) (* arc libre :
                                 on parcourt son représentant conforme *)
11                           let improve = Delta[x]+1 in
12                           if improve<Delta[y]
13                              then Delta[y] := improve;
14                                   (* p[y] := x : inutile *)
15                                   push_opt(work,y);
16               for each y ∈ pred(x) do
17                        if f(y,x)>0 (* arc passant :
                                 on parcourt son représentant non conforme *)
18                           let improve = Delta[x]+1 in
19                           if improve<Delta[y]
20                              then Delta[y] := improve;
21                                   (* p[y] := x : inutile *)
22                                   push_opt(work,y);
23     return Delta;
```

TABLE 19 – L'initialisation de la distance estimée ; les opérations `empty`, `push`, `pop`, `nonempty` sont des opérations de files (FIFO)

hors de T, $\mathtt{d}[x] \geq \mathtt{n}$ ». Ceci n'aura pas d'importance dans cette section, mais en aura au lemme 9.71, lorsque nous examinerons la correction de l'algorithme de Goldberg-Tarjan et Karzanov.

Cependant, on peut remarquer que les conditions $\mathtt{improve} < \mathtt{Delta}[\mathtt{y}]$ des lignes 12 et 19 sont vraies si et seulement si $\mathtt{Delta}[\mathtt{y}] = +\infty$, puisque chaque sommet n'est visité qu'une fois, ce qui permettrait de simplifier encore un peu l'algorithme d'initialisation.

Dans tous les cas, la complexité de l'algorithme d'initialisation de la table 19 est en $O(m+n)$. Ceci reste le cas même si on ajoute le temps d'un précalcul des listes de prédécesseurs `pred` en fonction du graphe $(\mathtt{S}, \mathtt{succ})$ exprimé en fonction de sa seule liste de successeurs.

Nous passons maintenant à l'étape 2(a) de l'algorithme de Dinic-Edmonds-Karp : $f := f.\mu$ (amélioration du flot). Ceci se code de façon immédiate par les formules données au lemme 9.17, mais l'on doit surtout vérifier que ceci n'entraîne *aucun* changement à effectuer sur la distance estimée Δ. C'est l'une des raisons pour lesquelles on utilise des distances estimées plutôt que des distances exactes : le graphe d'écart étant modifié lors de l'amélioration du flot, les distances exactes changent aussi, mais une distance estimée n'a pas à être mise à jour lors de cette étape. Voici le résultat qui le justifie ; on notera qu'il ne s'applique que si le chemin μ n'est pas seulement améliorant, mais admissible.

Lemme 9.26 *Soit f un flot sur un réseau (G, s, p), et Δ une distance estimée relative à f. Pour tout chemin admissible μ, Δ est encore une distance estimée relative à $f.\mu$.*

Démonstration. D'abord, on a toujours $\Delta(p) = 0$. Ensuite, pour tous les arcs $x \to_f y$ de $G_{f.\mu}$ qui étaient déjà dans G, on a $\Delta(x) \leq \Delta(y) + 1$, car Δ est une distance estimée sur G_f. (On ignore leurs coûts, qui n'ont rien à voir avec la notion de distance.) Il nous reste à examiner les nouveaux arcs, c'est-à-dire ceux présents dans $G_{f.\mu}$ mais pas dans G_f.

Or $G_{f.\mu}$ est obtenu à partir de G_f en effaçant certains arcs de μ, à savoir les arcs $x \to_f y$ tels que $r_f(x \to_f y) = \alpha(\mu)$, c'est-à-dire de coût minimum parmi les arcs du chemin μ, et en ajoutant possiblement leurs conjoints, s'ils existent. (Voir la remarque 9.12 pour la notion d'arc conjoint.) Les seuls nouveaux arcs sont donc ces arcs conjoints, nécessairement de la forme $y \to_{f.\mu} x$, c'est-à-dire avec x et y échangés. On doit vérifier que $\Delta(y) \leq \Delta(x) + 1$. Mais μ est un chemin *admissible*, donc $\Delta(x) = \Delta(y) + 1$. On en déduit $\Delta(y) = \Delta(x) - 1 \leq \Delta(x) + 1$. □

Le lemme suivant montre qu'on ne pourra effectuer l'étape d'amélioration de flot qu'au plus m fois.

Lemme 9.27 *Soit f un flot sur un réseau (G, s, p), et Δ une distance estimée relative à f. Pour tout chemin admissible μ, l'ensemble des arcs du graphe d'admissibilité $G_{f.\mu,\Delta}$ est strictement inclus dans l'ensemble des arcs du graphe d'admissibilité $G_{f,\Delta}$.*

Démonstration. $G_{f.\mu}$ est obtenu à partir de G_f en effaçant certains arcs de μ, à savoir les arcs $x \to_f y$ tels que $r_f(x \to_f y) = \alpha(\mu)$, c'est-à-dire de coût minimum parmi les arcs du chemin μ, et en ajoutant possiblement leurs conjoints, s'ils existent.

Or ces conjoints, $y \to_{f.\mu} x$, ne sont *pas admissibles* : comme $x \to_f y$ est admissible, on a $\Delta(x) = \Delta(y)+1$, donc $\Delta(y) \neq \Delta(x)+1$. Il s'ensuit qu'aucun arc ajouté n'est admissible, c'est-à-dire qu'aucun de fait partie du nouveau graphe d'admissibilité $G_{f.\mu,\Delta}$.

De plus, il y a au moins un arc supprimé entre $G_{f,\Delta}$ et $G_{f.\mu,\Delta}$, à savoir n'importe quel arc $x \to_f y$ sur le chemin μ tel que $r_f(x \to_f y) = \alpha(\mu)$. □

Pour ce qui est de l'étape 2(b) (mise à jour de Δ), une approche naïve serait de recalculer les distances exactes, en utilisant l'algorithme de la table 19 par exemple. Ce n'est pas nécessaire : on peut le mettre à jour par des corrections locales, en modifiant juste la distance partiel d'un seul sommet x. Nous avons besoin des notions suivantes.

Définition 9.28 *Soit f un flot sur un réseau (G, s, p), et Δ une distance estimée relative à f. Un* chemin admissible partiel *est un chemin $s = x_0 \to_f x_1 \to_f \cdots \to_f x_k = x$ composé d'arcs admissibles. Il est* bloqué *si et seulement si $x \neq p$ et pour tout arc $x \to_f y$ du graphe d'écart G_f, $\Delta(x) \neq \Delta(y) + 1$.*

chemin admissible partiel

bloqué

Les points 2 et 3 de la remarque 9.24 s'appliquent aussi aux chemins admissibles *partiels* μ : si $\Delta(s) < n$, alors $\Delta(x) < n$ pour tout sommet x de μ, et tout chemin admissible partiel est élémentaire.

Un chemin admissible partiel est la même chose qu'un chemin admissible, à ceci près qu'on ne demande pas que son extrémité x soit égale à p. Il est bloqué si et seulement si ($x \neq p$ et) on ne peut pas l'étendre en un chemin admissible partiel plus long. Dans ce cas, on mettra à jour Δ en la fonction $\mu.\Delta$ définie comme suit.

Définition 9.29 *Soit f un flot sur un réseau (G, s, p), et Δ une distance estimée relative à f. Pour tout chemin admissible partiel bloqué μ: $s \to_f^* x$, on pose $\mu.\Delta$ la fonction qui :*

1. *à x associe $\min\{\Delta(y) + 1 \mid y$ successeur de x dans $G_f\}$ si x a au moins un successeur dans G_f, n sinon.*
2. *à tout autre sommet z associe $\Delta(z)$.*

Ceci reste une distance estimée, et de plus une qui améliore Δ dans le sens expliqué ci-dessous.

Lemme 9.30 *Soit f un flot sur un réseau (G, s, p), et Δ une distance estimée relative à f. Pour tout chemin admissible partiel bloqué μ: $s \to_f^* x$, $\mu.\Delta$ est une distance estimée. De plus, $(\mu.\Delta)(y) \geq \Delta(y)$ pour tout sommet y, l'inégalité étant stricte si $y = x$.*

Démonstration. Comme μ est bloqué, $(\mu.\Delta)(x) > \Delta(x)$. En effet, sinon, on aurait $(\mu.\Delta)(x) \leq \Delta(x)$. Or $\Delta(x)$ est inférieur ou égal à la longueur du plus petit chemin de x à p dans G_f par le lemme 9.21, donc à $n-1$. L'argument est le même que pour le corollaire 9.22. Si x n'avait aucun successeur dans G_f, $(\mu.\Delta)(x)$ vaudrait n, or on vient de voir qu'il est inférieur ou égal à $n-1$. Donc x a au moins un successeur dans G_f. La définition de $(\mu.\Delta)(x)$ dans le cas où x a au moins un successeur dans G_f et l'inégalité $(\mu.\Delta)(x) \leq \Delta(x)$ nous donne maintenant l'existence d'un successeur y de x dans G_f tel que $\Delta(y)+1 \leq \Delta(x)$. Mais comme Δ est une distance estimée, $\Delta(x) \leq \Delta(y)+1$, et cette inégalité est stricte car μ est bloqué. Donc $\Delta(y)+1 \leq \Delta(x) < \Delta(y)+1$, ce qui est impossible.

Le fait que $(\mu.\Delta)(y) \geq \Delta(y)$ pour tout sommet y est maintenant évident.

Montrons maintenant que $\mu.\Delta$ est une distance estimée. D'abord, $(\mu.\Delta)(p) = \Delta(p) = 0$, puisque $x \neq p$, par définition d'un chemin bloqué. Ensuite, pour tout arc $y \to_f z$ dans G_f, on considère plusieurs cas.

- Si ni y ni z n'est égal à x, alors $(\mu.\Delta)(y) = \Delta(y) \leq \Delta(z)+1 = (\mu.\Delta)(z)+1$.
- Si $y = x$, alors $z \neq x$ parce qu'un réseau ne contient pas de boucle. On a alors $(\mu.\Delta)(y) = (\mu.\Delta)(x) \leq \Delta(z)+1$, par définition de $\mu.\Delta$.
- Si $y \neq x$ et $z = x$, alors $(\mu.\Delta)(y) = \Delta(y)$. Ceci est inférieur ou égal à $\Delta(z)+1$ puisque Δ est une distance estimée, c'est-à-dire à $\Delta(x)+1$. Or nous avons vu au début de cette démonstration que $\Delta(x) < (\mu.\Delta)(x)$. Donc $(\mu.\Delta)(y) < (\mu.\Delta)(x)+1 = (\mu.\Delta)(z)+1$. □

Notons sous forme d'un système de transition les effets des étapes 2(a) (amélioration du flot, $f := f.\mu$) et 2(b) (mise à jour de Δ. Nous en profitons pour raffiner la description abstraite de l'algorithme de Dinic-Edmonds-Karp en décrivant un peu plus explicitement comme les chemins admissibles sont découverts. Les configurations sont des triplets (f, Δ, μ) formés d'un flot f sur le réseau (G, s, p), d'une distance estimée Δ relative à f, et d'un chemin admissible partiel μ :

2(a) $(f, \Delta, \mu) \rightsquigarrow (f.\mu, \Delta, s)$, où μ est un chemin admissible dans G_f (la notation s dans $(f.\mu, \Delta, s)$ dénote le chemin admissible partiel réduit au seul sommet s) ;

2(b) $(f, \Delta, \mu) \rightsquigarrow (f, \mu.\Delta, \mathrm{chop}(\mu))$, où μ est un chemin admissible partiel bloqué dans G_f ; la fonction chop retourne μ si μ est de longueur 0, et sinon μ moins son dernier arc ;

2(c) $(f, \Delta, \mu) \rightsquigarrow (f, \Delta, \mathrm{extend}(\mu, y))$ où $m\colon s \to_f^* x$ n'est pas bloqué ; il existe alors un sommet y successeur de x dans G_f tel que $\Delta(x) = \Delta(y)+1$, et l'on note $\mathrm{extend}(\mu, y)$ le chemin (admissible partiel) obtenu en ajoutant l'arc $x \to_f y$ à la fin de μ.

Ces règles ne sont appliquées qu'à partir de configurations (f, Δ, μ) telles que $\Delta(s) < n$. La raison d'être de l'utilisation de la fonction chop dans les transitions

de type **2(b)** est de garantir que $\mathrm{chop}(\mu)$ est toujours un chemin admissible partiel, malgré l'augmentation de valeur de la distance estimée de son extrémité.

Lemme 9.31 *Si μ est un chemin admissible partiel bloqué, relativement à une fonction distance estimée Δ, alors* $\mathrm{chop}(\mu)$ *est un chemin admissible partiel relativement à $\mu.\Delta$.*

Démonstration. Soit $\mu\colon s = x_0 \to x_1 \to \cdots x_k$. Si $k = 0$, alors $\mathrm{chop}(\mu) = \mu$ est le chemin vide, qui est un chemin admissible partiel. Supposons donc $k \geq 1$. Comme μ est admissible, on $\Delta(s) = \Delta(x_i) + i$ pour tout $i \in \{0, 1, \cdots, k\}$, en particulier les sommets x_i sont distincts deux à deux. Or $\mu.\Delta(z) = \Delta(z)$ pour tout sommet z différent de x_k, par définition, et $\mu.\Delta(x_{i-1}) = \mu.\Delta(x_i) + 1$ pour tout $i \in \{1, \cdots, k-1\}$. Il s'ensuite que $\mathrm{chop}(\mu)\colon s = x_0 \to x_1 \to \cdots x_{k-1}$ est admissible relativement à $\mu.\Delta$. □

Chaque transition de type `2(b)` augmente $\sum_{x\in\mu} \Delta(x)$ d'au moins 1, autrement dit $\sum_{x\in\mu}(\mu.\Delta)(x) \geq \sum_{x\in\mu} \Delta(x) + 1$, où les sommes s'étendent sur tous les sommets du chemin μ. Ceci est une conséquence du lemme 9.30. Cette même quantité $\sum_{x\in\mu} \Delta(x)$ est inchangée par les transitions de type **2(a)**, et ne peut qu'augmenter par les transitions de type **2 (c)**. De plus, nous prétendons qu'elle est majorée par $n(n-1)$ (avant le déclenchement de chaque transition) : rappelons que les transitions **2(a)**, **2(b)** et **2(c)** ne sont appliquées que si $\Delta(s) < n$. Alors pour tout sommet x de μ, on a $\Delta(x) \leq \Delta(s) < n$, puisque tous les arcs de μ sont admissibles ; en fait, si k est la longueur du préfixe du chemin μ de s à x, $\Delta(x) = \Delta(s) - k$. Mais ceci implique que $\sum_{x\in\mu} \Delta(x) \leq n(n-1)$.

Donc on ne peut appliquer les transitions de type **2(b)** qu'au plus $n(n-1)$ fois. D'autre part, les transitions de type **2(a)** diminuent strictement la taille du graphe d'admissibilité $G_{f,\Delta}$ par le lemme 9.27, et ne peuvent donc s'appliquer qu'au plus m fois entre deux transitions de type **2(b)** (ou avant la première transition de type **2(b)**, ou après la dernière). Finalement, les transitions de type **2(c)**, qui étendent μ, ne peuvent s'appliquer qu'au plus n fois consécutivement.

On en déduit que les transitions de type **2(a)**, **2(b)** ou **2(c)** ne peuvent s'appliquer qu'un nombre fini de fois. Explicitement, au plus $n(n-1)$ applications de **2(b)**, entre chacune au plus m applications de **2(a)**, et entre chacune de ces dernières au plus n applications de **2(c)**. Ceci forme un total de $O(n^3m)$ transitions. On verra plus loin que c'est une estimation grossière. Elle l'est d'autant plus que les transitions ne sont *pas* en temps constant, et que la complexité estimé par la méthode ci-dessus est au-delà de l'ordre de n^3m.

En attendant, nous obtenons le résultat suivant.

Théorème 9.32 (Max-flow min-cut, Ford-Fulkerson) *Pour tout réseau (G, s, p),* $\max_{f \text{ flot}} \mathrm{val}(f) = \min_{E \text{ coupe}} \mathrm{b}(E)$.

Démonstration. L'inégalité $\leq$ est conséquence du lemme 9.8. Réciproquement, on a vu que l'algorithme abstrait de Dinic-Edmonds-Karp termine. À la fin de

l'algorithme, $\Delta(s) \geq n$, donc le flot f obtenu à la dernière configuration est un flot de valeur maximum, égale à $\mathrm{b}(E)$ pour une certain coupe E, par le corollaire 9.22. □

9.7 La complexité de l'algorithme de Dinic-Edmonds-Karp

Il ne nous reste qu'à raffiner l'analyse de complexité de l'algorithme de Dinic-Edmonds-Karp.

Lemme 9.33 *Soit $(f_0, \Delta_0, \mu_0) \rightsquigarrow (f_1, \Delta_1, \mu_1) \rightsquigarrow \cdots \rightsquigarrow (f_k, \Delta_k, \mu_k)$ une suite de transitions **2(a)**, **2(b)** ou **2(c)**, portant sur des configurations (f_i, Δ_i, μ_i) telles que f_i est un flot du réseau (G, s, p), Δ_i est une distance estimée relative à f_i, et μ_i est un chemin admissible partiel.*

Un couple (x, y) de sommets est un arc effacé *à la transition i si et seulement* arc effacé *$x \to_{f_{i-1}} y$ est un arc de $G_{f_{i-1}}$ mais $x \to_{f_i} y$ n'est pas un arc de G_{f_i}. Un couple (x, y) de sommets est un* arc ajouté *à la transition i si et seulement $x \to_{f_{i-1}} y$* arc ajouté *n'est pas un arc de $G_{f_{i-1}}$ mais $x \to_{f_i} y$ est un arc de G_{f_i}.*

Pour tout couple de sommets (x, y), si $x \to y$ est effacé ou ajouté à la transition i, alors :

1. *la transition $(f_{i-1}, \Delta_{i-1}, \mu_{i-1}) \rightsquigarrow (f_i, \Delta_i, \mu_i)$ est de type **2(a)** ;*
2. *$\Delta_i = \Delta_{i-1}$.*

Si $x \to y$ est un arc effacé à la transition i, alors :

3. *$x \to_{f_{i-1}} y$ est un arc du chemin μ_{i-1} ;*
4. *$\Delta_i(x) = \Delta_i(y) + 1$.*

Si $x \to y$ est un arc ajouté à la transition i, alors :

5. *le conjoint $y \to_{f_{i-1}} x$ est un arc du chemin μ_{i-1} ;*
6. *$\Delta_i(y) = \Delta_i(x) + 1$.*

Démonstration. 1. Les transitions de type **2(b)** et **2(c)** n'effacent ni n'ajoutent d'arcs, car elles laissent le flot inchangé, donc le graphe d'écart inchangé.

2. Par définition des transitions de type **2(a)**.

3. Par définition de $f_i = f_{i-1}.\mu_{i-1}$, on a $f_i(e) = f_{i-1}(e)$ pour tout arc de $G_{f_{i-1}}$ qui n'est pas sur le chemin μ_{i-1}. En particulier, aucun arc e hors de μ_{i-1} n'est effacé à la transition i.

4. Par le point 3, l'arc effacé $x \to_{f_{i-1}} y$ fait partie du chemin μ_{i-1}, qui est un chemin admissible partiel. Donc $\Delta_{i-1}(x) = \Delta_{i-1}(y) + 1$. Or $\Delta_{i-1} = \Delta_i$ par le point 2.

5. Supposons que $x \to y$ est ajouté à la transition i. Si $x \to_{f_i} y$ est le représentant conforme d'un arc $x \to y$ de G, on avait donc $f_{i-1}(x, y) = c_{\max}(x, y)$ (puisqu'il n'y avait pas d'arc $x \to_{f_{i-1}} y$ dans $G_{f_{i-1}}$) et $f_i(x, y) < c_{\max}(x, y)$. Ceci implique $f_{i-1}(x, y) > 0$, puisque $f_{i-1}(x, y) = c_{\max}(x, y) > f_i(x, y) \geq 0$. Donc le

conjoint était un arc $y \to_{f_{i-1}} x$ de $G_{f_{i-1}}$. Comme au point 3, il s'ensuit que ce conjoint est sur le chemin μ_{i-1}.

Si $x \to_{f_i} y$ est le représentant non conforme d'un arc $y \to x$ de G, on avait à la place $f_{i-1}(y,x) = 0$ (car il n'y avait pas d'arc $x \to_{f_{i-1}} y$ dans $G_{f_{i-1}}$) et $f_i(y,x) > 0$. Donc $f_{i-1}(y,x) < c_{\max}(y,x)$, puisque $f_{i-1}(y,x) = 0 < f_i(y,x) \leq c_{\max}(y,x)$. Donc le conjoint était un arc $y \to_{f_{i-1}} x$ de $G_{f_{i-1}}$, qui était donc sur le chemin μ_{i-1}.

6. Par le point 5, le conjoint $y \to_{f_{i-1}} x$ de l'arc ajouté $x \to_{f_i} y$ fait partie du chemin μ. Celui-ci étant admissible, on a $\Delta_{i-1}(y) = \Delta_{i-1}(x) + 1$. Or $\Delta_{i-1} = \Delta_i$ par le point 2. □

Lemme 9.34 *Soit $(f_0, \Delta_0, \mu_0) \rightsquigarrow (f_1, \Delta_1, \mu_1) \rightsquigarrow \cdots \rightsquigarrow (f_k, \Delta_k, \mu_k)$ une suite de transitions comme au lemme 9.33, avec $\Delta_{i-1}(s) < n$ pour tout $i \in \{1, \cdots, k\}$. Pour tout couple de sommets (x, y), l'arc $x \to y$ ne peut être effacé qu'à au plus $\lceil \frac{n}{2} \rceil \leq \frac{n+1}{2}$ des k transitions.*

Démonstration. Considérons deux effacements successifs de $x \to y$, autrement dit supposons que $x \to y$ est un arc effacé à la transition i, de nouveau effacé à une transition $\ell > i$, et que $x \to y$ ne soit effacé à aucune des transitions $i+1$, $\ldots$, $\ell - 1$. Nécessairement, $x \to y$ est ajouté à au moins une transition j, avec $i < j < \ell$. Par le lemme 9.33, points 4 et 6, on a :

$$\begin{aligned}
\Delta_i(x) &= \Delta_i(y) + 1 \\
\Delta_j(y) &= \Delta_j(x) + 1 \\
\Delta_\ell(x) &= \Delta_\ell(y) + 1.
\end{aligned}$$

Or $\Delta_i \leq \Delta_j \leq \Delta_\ell$ par le lemme 9.30. Donc :

$$\begin{aligned}
\Delta_i(y) &= \Delta_i(x) - 1 \\
&\leq \Delta_j(x) - 1 \\
&= \Delta_j(y) - 2 \\
&\leq \Delta_\ell(y) - 2.
\end{aligned}$$

Autrement dit, la valeur de la distance estimée en y augmente d'au moins 2 entre deux effacements successifs de $x \to y$. Or par hypothèse les transitions ne sont effectués que si la valeur de la distance estimée $\Delta_{i-1}(s)$ en s est strictement inférieure à n. Par le point 3 du lemme 9.33, à chaque transition i où l'arc $x \to y$ est effacé, $x \to_{f_{i-1}} y$ est un arc du chemin μ_{i-1}, donc y est un somme de ce chemin. Mais par définition d'un chemin admissible partiel, ceci implique que $\Delta_{i-1}(y) \leq \Delta_{i-1}(s) < n$.

En résumé, la la valeur de la distance estimée en y augmente d'au moins 2 entre deux effacements successifs de $x \to y$, et ne peut pas dépasser $n - 1$ au déclenchement du dernier effacement de $x \to y$. Donc $x \to y$ ne peut être effacé qu'au plus $\lceil \frac{n}{2} \rceil$ fois. □

Nous pouvons améliorer notre estimation précédente du nombre de fois où l'on peut appliquer une transition de type **2(a)**. Rappelons que notre estimation précédente était d'au plus m fois entre deux transitions de type **2(b)**, lesquelles sont en nombre au plus $n(n-1)$. Notre estimation précédente majorait donc le nombre d'applications de transitions de type **2(a)** par $n(n-1)m$. Le lemme suivant fournit un meilleur majorant.

Lemme 9.35 *Soit* $(f_0, \Delta_0, \mu_0) \rightsquigarrow (f_1, \Delta_1, \mu_1) \rightsquigarrow \cdots \rightsquigarrow (f_k, \Delta_k, \mu_k)$ *une suite de transitions comme au lemme 9.33, avec* $\Delta_{i-1}(s) < n$ *pour tout* $i \in \{1, \cdots, k\}$. *Alors au plus* $m(n+1)$ *de ces transitions sont de type* `2(a)`.

Démonstration. Tout transition de type **2(a)** efface au moins un arc $x \rightarrow_f y$, qui est celui (ou l'un de ceux) du chemin admissible μ tel que $r_f(x,y) = \alpha(\mu)$. Ces arcs du graphe d'écart peuvent être conformes ou non conformes, c'est-à-dire être un représentant d'un arc $x \rightarrow y$ ou $y \rightarrow x$ de G. Les couples (x,y) tels qu'on peut effacer $x \rightarrow y$ sont donc en nombre majoré par $2m$ (pas m !). On ne peut effacer chacun qu'au plus $(n+1)/2$ fois dans la suite de transitions données, d'où le résultat. □

Nous avions vu qu'il ne pouvait y avoir qu'au plus $n(n-1)$ de transitions de type **2(b)** lors d'une exécution. On peut raffiner ceci comme suit.

Lemme 9.36 *Soit* $(f_0, \Delta_0, \mu_0) \rightsquigarrow (f_1, \Delta_1, \mu_1) \rightsquigarrow \cdots \rightsquigarrow (f_k, \Delta_k, \mu_k)$ *une suite de transitions comme au lemme 9.33, avec* $\Delta_{i-1}(s) < n$ *pour tout* $i \in \{1, \cdots, k\}$. *Pour tout sommet* x *de* G, *il n'y qu'au plus* n *transitions* $(f_{i-1}, \Delta_{i-1}, \mu_{i-1}) \rightsquigarrow (f_i, \Delta_i, \mu_i)$ *de type* ***2(b)*** *telles que* $\Delta_{i-1}(x) < \Delta_i(x)$.

Démonstration. En effet, si $\Delta_{i-1}(x) < \Delta_i(x)$, alors x est l'extrémité d'un chemin admissible partiel μ_{i-1}, par définition des transitions de type **2(b)**, donc $\Delta_{i-1}(x) \leq \Delta_{i_1}(s) < n$. □

Une analyse similaire majore le nombre de transitions de type **2(c)** par sommet pouvant apparaître comme extrémité de μ_{i-1}.

Lemme 9.37 *Soit* $(f_0, \Delta_0, \mu_0) \rightsquigarrow (f_1, \Delta_1, \mu_1) \rightsquigarrow \cdots \rightsquigarrow (f_k, \Delta_k, \mu_k)$ *une suite de transitions comme au lemme 9.33, avec* $\Delta_{i-1}(s) < n$ *pour tout* $i \in \{1, \cdots, k\}$. *La transition* $(f_{i-1}, \Delta_{i-1}, \mu_{i-1} \rightsquigarrow (f_i, \Delta_i, \mu_i)$ *est la* transition i. *Disons que :* transition i

- *un sommet* z apparaît à la transition i *si et seulement si on le trouve sur le chemin* μ_i *mais pas sur le chemin* μ_{i-1}, *ou bien si* $z = s$ *et la transition* i *est de type* ***2(a)*** ; apparaît à la transition i
- *un sommet* z *il* disparaît à la transition i *si et seulement si on le trouve sur le chemin* μ_{i-1} *et pas sur le chemin* μ_i, *ou bien si* $z = s$ *et la transition* i *est de type* ***2(a)*** ; disparaît à la transition i
- *un* segment *d'un sommet* z *est un intervalle entier* $[j, \ell]$ *avec* $1 \leq j < \ell \leq k+1$, *tel que* z *apparaît à la transition* j, *et* ℓ *est le plus petit entier* segment

$\ell \in \{j+1, \cdots, k\}$ *tel que* z *disparaît à la transition* ℓ*, ou bien* $k+1$ *si un tel entier n'existe pas ;*

— *un couple de sommets* (x, y) *est* candidat à l'extension à la transition i *si* x *est l'extrémité du chemin* μ_{i-1}*,* y *est un successeur de* x *dans* $G_{f_{i-1}}$ *(donc* $y \in \texttt{succ}(x) \cup \texttt{pred}(x)$*), et* $\Delta_{i-1}(x) = \Delta_{i-1}(y) + 1$*, autrement dit si les conditions d'applicabilité d'une transition de type* ***2(c)*** *sont vérifiées.*

candidat à l'extension à la transition i

Alors :

1. *Pour tout sommet* x *de* G*, pour tout segment* $[j, \ell]$ *de* x*, pour tout sommet* y *de* G*, l'une des trois possibilités suivantes se présente, de façon exclusive :*
   (a) (x, y) *est candidat à l'extension à toute transition de* $j+1, \ldots, \ell-1$ *;*
   (b) (x, y) *n'est candidat à l'extension à aucune transition de* $j+1, \ldots, \ell-1$ *;*
   (c) *il existe un unique entier* $p \in \{j+1, \cdots, \ell-1\}$ *tel que* y *apparaît à la transition* p*,* (x, y) *est candidat à l'extension à toute transition* $j+1, \ldots, p-1$ *et ne l'est à aucune transition* $p+1, \ldots, \ell-1$*.*
2. *Tout sommet* x *disparaît à au plus* $m(n+1)+n$ *transitions.*
3. *Tout sommet* x *a au plus* $(m+1)(n+1)$ *segments. , il y a au plus* $(m+1)(n+1)(\#\texttt{succ}(x) + \#\texttt{pred}(x))$ *transitions* i *de type* ***2(c)*** *telles que l'extrémité de* μ_{i-1} *soit le sommet* x*.*

Démonstration. Le point clé est que tout sommet qui disparaît via une transition de type **2(c)** provoque une augmentation stricte de sa distance estimée. Explicitement, supposons que z disparaisse à la transition i, et qu'il s'agisse d'une transition de type **2(c)**. Alors z est l'extrémité de μ_{i-1}. Par le lemme 9.30, $\Delta_i(z) = (\mu_{i-1}.\Delta_{i-1})(z) > \Delta_{i-1}(z)$,.

Notons ensuite que dans un segment $[j, \ell]$, aucun des transitions $j+1, \ldots, \ell-1$ ne peut être de type **2(a)**. En effet, par définition une telle transition fait disparaître tous les sommets (et fait réapparaître s, qui est le seul sommet qui peut apparaître à une transition où il disparaît).

1. Comme x ne disparaît à aucune transition $j+1, \ldots, \ell-1$, sa distance estimée ne change pas, autrement dit $\Delta_m(x) = \Delta_j(x)$ pour tout $m \in \{j+1, \cdots, \ell-1\}$. Pour un sommet fixé y, (x, y) ne peut être candidat à l'extension à une transition m que si $\Delta_{m-1}(x) = \Delta_{m-1}(y) + 1$. Or la distance estimée de y ne peut qu'augmenter ou rester égale : par le lemme 9.30 en cas de transition de type **2(c)**, auquel cas l'augmentation est stricte si et seulement y est l'extrémité de μ_{i-1}, et donc disparaît, ou bien parce que les distances estimées ne changent pas aux cours des transitions de type **2(b)**.

Donc l'ensemble des numéros $m \in \{j+1, \cdots, \ell-1\}$ tel que $\Delta_{m-1}(x) = \Delta_{m-1}(y)+1$ est soit vide, soit l'ensemble tout entier, soit un sous-intervalle gauche $\{j+1, \cdots, p\}$ avec $j < p < \ell$.

```
   fun dinic-extend (musuccx, mupredx, f, cmax, Delta, x) =
1      while musuccx≠[] do
2            let y::rest=musuccx in
3                (musuccx := rest; (* on dépile y de musucc *)
4                 if f(x,y)<cmax(x,y) and Delta[x]=Delta[y]+1
5                    then return y, true); (* sommet successeur y,
                                              arc conforme *)
6      while mupredx≠[] do
7            let y::rest=mupredx in
8                (mupredx := rest; (* on dépile y de mupred *)
9                 if f(y,x)>0 and Delta[x]=Delta[y]+1
10                   then return y, false; (* sommet successeur y,
                                               arc non conforme *)
11     raise NotFound;
```

TABLE 20 – La recherche d'un sommet permettant d'étendre un chemin partiel admissible

Comme aucune transition $j+1, \ldots, \ell-1$ n'est de type **2(a)**, le graphe d'écart reste inchangé, et donc la condition « y est un successeur de x dans $G_{f_{m-1}}$ » est fausse pour tout $m \in \{j+1, \cdots, \ell-1\}$, ou vraie pour tout $m \in \{j+1, \cdots, \ell-1\}$.

2. Le nombre de transitions où x disparaît est majoré par le nombre de transitions de type **2(a)**, où tous les sommets disparaissent (au plus $m(n+1)$, par le lemme 9.35), plus le nombre de transitions de type **2(c)** où x disparaît. À toute transition i de type **2(c)** où x disparaît, on a $\Delta_i(x) > \Delta_{i-1}(x)$. Or, comme μ_{i-1} est un chemin admissible partiel, $\Delta_{i-1}(x)$ est égal à $\Delta_{i-1}(s)$ moins la longueur de μ_{i-1}, et est donc inférieur ou égal à $\Delta_{i-1}(s) < n$. Il ne peut donc y avoir qu'au plus n transitions de type **2(c)** où x disparaît.

3. Il y a au plus autant de segments de x que de nombres de transitions où x disparaît, plus un ; c'est-à-dire $(m+1)(n+1)$ par le point 2. Pour chacun de ces segments $[j, \ell]$, le nombre N de transitions m de type **2(c)** telles que μ_{m-1} soit d'extrémité x est majoré par le nombre de sommets y tels que (x, y) soit candidat à l'extension à l'une des transitions $j+1, \ldots, \ell-1$, puisque chacun de des sommets y ne peut apparaître qu'au plus à une des transitions $j+1, \ldots, \ell-1$ par le point 1. Comme chacun de ces sommets y est un successeur ou un prédécesseur de x dans G, $N \leq \#\texttt{succ}(x) + \#\texttt{pred}(x)$. □

La table 25 donne l'algorithme de Dinic-Edmonds-Karp final, et utilise plusieurs fonctions auxiliaires (tables 20 et suivantes), que nous commenterons plus bas. Il s'agit d'une réalisation quasiment directe des transitions de type **2(a)**, **2(b)** et **2(c)**. On maintient comme principaux invariants que `f` est un flot de $(\texttt{G}, \texttt{s}, \texttt{p})$, que `Delta` est une distance estimée relative à `f`, et que `mu` (avec `nmu`) représente

```
   fun rf (f, cmax, x, y, est-conforme) =
1      if est-conforme
2         then return cmax(x,y)-f(x,y) (* conforme *)
3      else return f(y,x) ; (* non conforme ; attention au renversement de l'arc ! *)
```

TABLE 21 – Le calcul de $r_f(x, y)$ pour un arc $x \to_f y$ du graphe d'écart

```
   fun alpha (f, cmax, mu, conforme, nmu) =
1      a := rf (f, cmax, mu[0], mu[1], conforme[0]) ;
2      for i=1 to nmu-1 do
3          let new-a = rf (f, cmax, mu[i], mu[i+1], conforme[i]) in
4              if new-a<a
5                 then a := new-a ; (* calcul du minimum *)
6      return a ;
```

TABLE 22 – Le calcul de $\alpha(\mu)$

```
   fun update-flow (f, cmax, mu, conforme, nmu) =
1      a := alpha (f, cmax, mu, conforme, nmu) ;
2      for i=0 to nmu-1 do
3          if conforme[i]
4             then f (mu[i], mu[i+1]) += alpha ; (* arc conforme *)
5          else f (mu[i+1], mu[i]) -= alpha ; (* arc non conforme ;
                                                 attention au renversement de l'arc ! *)
```

TABLE 23 – La mise à jour d'un flot par un chemin améliorant (lemme 9.17)

```
   fun min-delta (succ, pred, f, cmax, Delta, x) =
1      d := n ;
2      for each y ∈ succ(x) do
3               if f(x,y)<cmax(x,y) do
4                  d := min (d, Delta[y]+1) ; (* sommet successeur y, arc conforme *)
5      for each y ∈ pred(x) do
6               if f(y,x)>0 do
7                  d := min (d, Delta[y]+1) ; (* sommet successeur y, arc non conforme *)
8      return d ;
```

TABLE 24 – Le calcul de $\min\{\Delta(y) + 1 \mid y \text{ successeur de } x \text{ dans } G_f\}$ (ou n si l'ensemble est vide)

```
   fun dinic-edmonds-karp (G as (S, succ, pred), s, p, cmax, n) =
1      let f = new-flow(n) in for each x ∈ S do
2                                       for each y ∈ succ(x) do
3                                               f(x,y) := 0 ; (* initialisation du flot *)
4      Delta := init-delta (G, p, f, cmax, n) ; (* voir la table 19 *)
5      mu:=new(n); mu[0]:=s; musucc[0]:=succ(s); mupred[0]:=pred(s); nmu:=0 ;
              (* initialisation de μ *)
6      conforme := new(n) ;
7      while Delta[s]<n do
8            try (while mu[nmu]≠p do
9                         (mu[nmu+1], conforme[nmu] :=
                              dinic-extend (&musucc[nmu], &mupred[nmu],
                                            f, cmax, Delta, mu[nmu]) ;
10                        nmu := nmu+1 ; (* type 2(c), extension de μ *)
11                        musucc[nmu] := succ(mu[nmu]) ;
12                        mupred[nmu] := pred(mu[nmu])) ;
13                update-flow (f, cmax, mu, conforme, nmu) ; (* type 2(a) *)
14                mu[0] := s; nmu := 0) ; (* réinitialisation de μ *)
15           match NotFound => (* type 2(b), mise à jour de Δ *)
16                 (Delta[mu[nmu]] := min-delta (succ, pred, f,
                                                 cmax, Delta, mu[nmu]) ;
17                  if nmu≠0 then nmu := nmu-1 ;)
18     return f ;
```

TABLE 25 – L'algorithme de Dinic-Edmonds-Karp

```
   fun flow-val (G as (S, succ, pred), f, s) =
1      v := 0;
2      for each v ∈ succ(s) do
3               v := v+f(s,v);
4      for each v ∈ pred(s) do
5               v := v-f(v,s);
6      return v;
```

TABLE 26 – Le calcul de la valeur d'un flot $\mathrm{val}(f, \{s\})$

un chemin admissible partiel μ. On suppose μ représenté sous forme d'un tableau `mu` de longueur maximale `n`, alloué à la ligne 5, ses éléments $\texttt{mu}[0], \ldots, \texttt{mu}[\texttt{nmu}]$ étant les sommets successifs sur le chemin μ. La fonction `new-flow` à la ligne 1 est non spécifiée, et est supposée allouer une structure (par exemple de tableau $n \times n$) qui contiendra le flot `f`.

Le tableau de booléens `conforme` alloué à la ligne 6 répond à un problème subtil. Nous avons noté à la remarque 9.11 que le graphe d'écart est un *multi-graphe*, pas un graphe. Donc si $x \stackrel{\text{def}}{=} \texttt{mu}[i-1]$ et $y \stackrel{\text{def}}{=} \texttt{mu}[i]$, alors il peut y avoir *deux* arcs $x \to_f y$ dans le graphe d'écart G_f. Ceci ne nous a pas posé de problème jusqu'ici, parce qu'un chemin admissible ne peut contenir qu'un seul de ces deux arcs. Mais, dans une implémentation, nous devons savoir lequel : $\texttt{conforme}[i-1]$ vaudra `true` si c'est l'arc conforme qu'on y trouve, et `false` sinon.

Quant aux tableaux de listes `musucc` et `mupred`, ils contiennent pour chaque indice $i \in \{0, \cdots, \texttt{nmu}\}$, la liste des successeurs et des prédécesseurs de $\texttt{mu}[i]$ qui n'ont pas encore été explorés par `dinic-extend`. (Je vais commencer à être un peu moins formel à ce sujet.) Pour chaque sommet x ajouté au chemin μ, que ce soit à l'initialisation (ligne 5) ou après un appel à `dinic-extend` (lignes 11–12), on les initialise aux listes $\texttt{succ}(x)$ et $\texttt{pred}(x)$ de x dans G. Elles sont passées par référence à `dinic-extend` (le symbole `&`, venant du langage C) de sorte à éliminer les transitions déjà explorées. Le fait que ceci soit correct est dû au point 1 lemme 9.37 : les seuls sommets y à explorer pour appliquer une transition de type **2(c)** sur un chemin d'extrémité x sont leurs prédécesseurs ou leurs successeurs y, et on n'aura jamais à considérer le même y deux fois avant que x disparaisse, c'est-à-dire sur le même segment de x.

Les lignes 1–3 initialisent `f` au flot nul, en temps $O(m+n)$. La ligne 4 fait appel à la procédure d'initialisation déjà décrite à table 19, qui s'exécute aussi en temps $O(m+n)$. Les lignes 5–6 initialisent le chemin μ de sorte à ne contenir que s.

Remarque 9.38 *Dans un réseau, $2n-4 \leq m$. En effet, pour chacun des $n-2$ sommets x autres que s et p, il y a au moins un chemin $s \to^* u \to x$ et au moins un chemin $x \to v \to^* p$, donc au moins deux arcs $u \to x$ et $x \to v$; de plus, ils*

sont distincts car un réseau n'a pas de boucle. Donc on peut simplifier l'expression $O(m+n)$ de la complexité de l'initialisation de l'algorithme de Dinic-Edwards-Karp, et parler d'une complexité en $O(m)$. (Ce genre de raisonnement est faux sur un graphe arbitraire.)

Les lignes 7–17 implémentent les transitions **2(a)**, **2(b)** et **2(c)** tant que la distance estimée de `s` est strictement inférieure à `n`, le nombre de sommets du graphe `G` donné en entrée. La ligne 9 appelle une fonction `dinic-extend` (voir la table 20) qui tente de calculer une nouvel arc du graphe d'écart à ajouter à la fin du chemin μ pour implémenter une transition de type **2(c)**.

S'il n'y en a pas, il lance l'exception `NotFound`, qui est capturée à la ligne 15 et force la mise à jour de la distance estimée, par une transition de type **2(b)**. Le calcul de la nouvelle distance estimée (définition 9.29) est effectué par la fonction `min-delta` de la table 24, et la ligne 17 implémente l'opération chop des transitions de type **2(b)**.

Si en revanche les transitions de type **2(c)** effectuées aux lignes 9–10 finissent par produire un chemin admissible, c'est-à-dire lorsque la boucle `while` des lignes 8–10 termine sans lancer d'exception, une transition de type **2(a)** de mise à jour du flot est effectuée, en appelant la fonction auxiliaire `update-flow` de la table 23. Cette dernière fonction appelle la fonction `alpha` de la table 22, qui calcule $\alpha(\mu)$, laquelle appelle elle-même la fonction `rf` de la table 21, qui calcule les coûts $r_f(e)$ d'arcs e du graphe d'écart G_f.

Théorème 9.39 *L'algorithme de Dinic-Edmonds-Karp calcule correctement un flot de valeur maximum d'un réseau donné en entrée en temps $O(nm^2)$.*

Si l'on souhaite la *valeur* de ce flot f, on peut ensuite appeler la fonction `flow-val` sur le graphe G, le flot f, et la source s, voir la table 26 ; ceci s'exécute en temps $O(m)$.

Démonstration. D'abord, il est correct. Il calcule une suite de transitions $(f_0, \Delta_0, \mu_0) \rightsquigarrow (f_1, \Delta_1, \mu_1) \rightsquigarrow \cdots \rightsquigarrow (f_k, \Delta_k, \mu_k)$ une suite de transitions **2(a)**, **2(b)** ou **2(c)**, portant sur des configurations (f_i, Δ_i, μ_i) telles que f_i est un flot du réseau (G, s, p), Δ_i est une distance estimée relative à f_i, et μ_i est un chemin admissible partiel. S'il termine, alors (f_k, Δ_k, μ_k) est une configuration telle que $\Delta_k(s) \geq n$, donc f_k est un flot de valeur maximum par le corollaire 9.22, qu'il retourne.

On a vu que l'initialisation des lignes 1–6 prenait un temps $O(m)$ (voir la remarque 9.38).

Nous prétendons que la complexité cumulée des lignes 9–12 à travers l'exécution de tout l'algorithme (pas juste à l'intérieur d'un tour de la boucle des lignes 7–17) est en $O(nm^2)$. Pour chaque sommet fixé x, x a au plus $(m+1)(n+1)$ segments par le point 3 du lemme 9.37. Dans chaque segment $[j, \ell]$ de x, la temps pris par les différents appels à `dinic-extend` à la ligne 9 lorsque x est égal à

`mu`[`nmu`] est majoré par quelque chose de proportionnel à $\#\texttt{succ}(x) + \#\texttt{pred}(x)$ plus une constante, puisque x ne disparaît à aucune des transitions $j+1, \ldots, \ell-1$, et que donc les listes `musucc`$[nmu]$ et `mupred`$[nmu]$ (aux transitions où $x = \texttt{mu}[\texttt{nmu}]$) ne font que décroître. En sommant sur tous les sommets x, la complexité cumulée des lignes 9–12 est majorée par quelque chose de proportionnel à $\sum_{x\in S}(m+1)(n+1)(\#\texttt{succ}(x)+\#\texttt{pred}(x)+1) = (m+1)(n+1)(2m+1) = O(nm^2)$.

Par le lemme 9.35, on ne passe à la ligne 13 (transitions de type **2(a)**, amélioration du flot) qu'au plus $m(n+1)$ fois, et la mise à jour du flot prend un temps égal à la somme de :

— quelque chose de proportionnel à la longueur du chemin μ (dans la variable `mu` de la fonction α à la table 22, appelée à la ligne 1 de la fonction `update` − `flow` de la table 23) plus une constante pour calculer $\alpha(\mu)$, ce qui est majoré par une fonction affine de n, puisque μ est un chemin élémentaire ;

— un temps du même ordre pour les lignes 2–5 de la fonction `update` − `flow`.

Le temps cumulé par la ligne 13, et donc aussi par les lignes 13–14, de la table 25 est donc majoré par quelque chose de proportionnel à $m(n+1)(n+1)$, qui est donc en $O(n^2m)$.

Finalement, pour chaque sommet x tel que $x = \texttt{mu}[\texttt{nmu}]$ à la ligne 16, sa distance estimée est mise à jour par une transition de type **2(b)**, et ceci ne peut arriver qu'à au plus n transitions au cours de l'algorithme, par le lemme 9.36, pour chaque sommet x donné. Le temps pris pour calculer `min-delta` (voir la table 24) est majoré par une fonction affine de $\#\texttt{succ}(x) + \#\texttt{pred}(x)$, donc le temps cumulé par la ligne 16, et donc aussi par les lignes 16–17 est majoré par quelque chose de proportionnel à $\sum_{x\in S} n(\#\texttt{succ}(x)+\#\texttt{pred}(x)+1) = n(2m+1)$, qui est en $O(nm)$.

La complexité totale de l'algorithme de Dinic-Edmonds-Karp est donc en $O(m)$ (initialisation) plus $O(nm^2)$ (lignes 9–12) plus $O(n^2m)$ (lignes 13–14) plus $O(nm)$ (lignes 16–17). Le terme dominant est celui des lignes 9–12 (l'extension de chemin par des transitions de type **2(c)**), et la complexité finale est donc en $O(nm^2)$. □

Remarque 9.40 *Il est écrit dans le BBC (théorème 3.6, section 8.3.2, page 262) qu'un algorithme essentiellement identique à* ***`dinic-edwards-karp`*** *(sans mention explicite de la nécessité de maintenir des tableaux de listes* `musucc` *et* `mupred` *pour éviter des recalculs inutiles lors des extensions de chemins) s'exécute en temps $O(n^2m)$, ce qui est meilleur que la complexité $O(nm^2)$ annoncée au théorème 9.39. Mais l'analyse du BBC semble considérer implicitement que l'étape d'extension de chemins (*`dinic-extend`* pour nous) s'effectue en temps constant, ce qui est erroné.*

Remarque 9.41 *L'analyse de complexité que nous avons faite de l'algorithme de Dinic-Edwards-Karp suppose que les opérations arithmétiques faites sur les*

coûts sont en temps constant. Voir la remarque 7.26 pour une discussion de cette question.

9.8 L'algorithme par échelonnement d'Edmonds-Karp et Dinic

Edmonds et Karp ont présenté un autre algorithme de calcul de flots maximums dans le même article que celui déjà cité (J. Edmonds, R. M. Karp, *Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems*, J. ACM 19(2) :248-264, 1972), qui a été découvert en parallèle par Dinic (E. A. Dinic, *An Efficient Algorithm for the Solution of the Generalized Set Representatives Problem*, Voprosy Kibernetiki, Proc. of the Seminar on Combinatorial Mathematics, Moscow, 1971, Scientific Council on the Complex Problem « Kibernetika », Akad. Nauk SSSR, pages 49–54, 1973 ; ce n'est pas le même article que celui de Dinic cité antérieurement).

Ce nouvel algorithme s'applique dans le cas particulier où les capacités maximales sont *entières*, ou de façon plus générale, toutes multiples entières d'un même réel strictement positif, ce qui est notamment le cas si les capacités maximales sont rationnelles.

Au lieu de chercher des chemins améliorants μ minimisant une distance estimée, ces algorithmes cherchent des chemins améliorants μ tels que $\alpha(\mu)$ soit le plus grand possible. Comme cherche le chemin maximisant $\alpha(\mu)$ est complexe, l'algorithme de se contenter de chercher des chemins tels que $\alpha(\mu) \geq 2^k$, puis tels que $\alpha(\mu) \geq 2^{k-1}$, etc.

Définition 9.42 *Soit f un flot sur un réseau (G, s, p). Pour tout $D > 0$, le* graphe d'écart tronqué *$G_f(D)$ est le sous-graphe couvrant de G_f formé des arcs e de coûts $r_f(e) \geq D$.*

graphe d'écart tronqué

Le lemme clé est le suivant.

Lemme 9.43 *Soit (G, s, p) un réseau à m arcs et $v_{\max}$ la valeur de son flot maximum. Soit f un flot sur (G, s, p), et $D > 0$. S'il n'existe pas de chemin améliorant dans $G_f(D)$, alors* $\mathrm{val}(f) \geq v_{\max} - mD$.

Démonstration. Soit E l'ensemble des sommets accessibles depuis s dans $G_f(D)$. Par hypothèse, $p \notin E$, et comme $s \in E$, E est une coupe. On a donc $\mathrm{val}(f) \leq v_{\max} \leq \mathrm{b}(E)$, par le lemme 9.8.

On peut évaluer $\mathrm{val}(f)$ à l'aide de n'importe quelle coupe (voir la définition 9.6, fondée sur le lemme 9.5), donc $\mathrm{val}(f) = \mathrm{out}(f, E) - \mathrm{in}(f, E)$.

Pour tout arc $e\colon u \in E \to w \notin E$, s'il a un représentant conforme $u \to_f w$, ce représentant conforme ne peut pas être dans $G_f(D)$, puisqu'alors w serait accessible depuis s dans $G_f(D)$ via u, ce qui n'est pas le cas puisque $w \notin E$ et E est l'ensemble des sommets accessibles depuis s dans $G_f(D)$. Donc $r_f(u \to_f w) < D$,

autrement dit $c_{\max}(u,w) - f(u,w) < D$. Cette inégalité est aussi vraie si e n'a pas de représentant conforme, autrement dit si e est saturé, auquel cas $c_{\max}(u,w) - f(u,w) = 0$. Donc $\mathrm{out}(f,E) = \sum_{u\in E \to w\notin E} f(u,w) \geq \sum_{u\in E\to w\notin E} c_{\max}(u,w) - out_E.D = \mathrm{out}(c_{\max},E) - out_E.D$, où out_E est le nombre d'arcs sortants de E.

De façon symétrique, pour tout arc $e\colon : v \notin E \to u \in E$, s'il a un représentant non conforme $u \to_f v$, il ne peut pas être dans $G_f(D)$, parce que E est l'ensemble des sommets accessibles depuis s dans $G_f(D)$. Donc $r_f(u \to_f v) < D$, autrement dit $f(v,u) < D$. Ceci est aussi vrai si e n'a pas de représentant non conforme, autrement dit si e est inerte, auquel cas $f(v,u) = 0$. Donc $\mathrm{in}(f,E) = \sum_{v\notin E\to u\in E} f(e) \leq in_E.D$, où in_E est le nombre d'arcs entrants de D.

On a donc :

$$\begin{aligned}\mathrm{val}(f) &= \mathrm{out}(f,E) - \mathrm{in}(f,E)\\ &\geq \mathrm{out}(c_{\max},E) - out_E.D - in_E.D\\ &= \mathrm{b}(E) - out_E.D - in_E.D\\ &\geq v_{\max} - m.D,\end{aligned}$$

puisque $out_E + in_E \leq m$. □

L'algorithme par échelonnement (« capacity scaling », en anglais) d'Edmonds-Karp et Dinic fonctionne comme suit :

1. Initialisation. On pose f égal au flot nul, et on calcule un entier $k \geq 1$ tel que $2^k \geq c_{\max}(e)$ pour tout arc e de G, le plus petit possible, en temps $O(n+m)$ (par un parcours de S puis des listes de successeurs de chaque sommet ; pour tout entier a, trouver le plus petit entier $k \geq 1$ tel que $2^k \geq a$ est supposé effectué en temps constant).
2. Tant que $k \geq 1$:
   (a) Tant qu'il existe un chemin améliorant μ dans $G_f(2^{k-1})$, $f := f.\mu$ (améliorer le flot) ;
   (b) décrémenter k.

L'invariant principal de cet algorithme, vrai au début de l'étape 2, est :

(Inv 1) $\mathrm{val}(f) \geq v_{\max} - m2^k$.

Il est vérifié après l'initialisation, var $v_{\max} \leq \sum_{u\to v} c_{\max}(u,v) \leq m2^k$, puisque k a été initialisé de sorte à ce que $c_{\max}(u,v) \leq 2^k$ pour tout arc $u \to v$ de G. Ensuite, on ne peut sortir de la boucle 2(a) que lorsqu'il n'existe plus de chemin de s à p dans $G_f(2^{k-1})$. Le lemme 9.43 nous dit alors que le flot ainsi obtenu satisfait $\mathrm{val}(f) \geq v_{\max} - m2^{k-1}$. Après la décrémentation en 2(b), on a donc $\mathrm{val}(f) \geq v_{\max} - m2^k$, ce qui rétablit l'invariant **(Inv 1)**.

Théorème 9.44 *Sur un réseau (G,s,p) à capacités maximums entières donné en entrée, l'algorithme par échelonnement d'Edmonds-Karp et Dinic retourne un flot maximum en temps $O(m^2 \max(1,\log C))$, où C est la capacité maximum des arcs de G.*

L'expression de la complexité est souvent exprimée sous la form $O(m^2 \log C)$, et ceci est valide pour C assez grand. Le cas $C = 1$ sera important à la section 9.9, et écrire $O(m^2 \log C)$ (soit 0) dans ce cas serait une complexité erronée.

Démonstration. À la fin de la dernière itération de la boucle externe (étape 2), k valait 1, donc l'étape 2(a) termine quand il n'existe plus de chemin améliorant dans $G_f(2^{k-1}) = G_f(1)$. Mais comme toutes les valeurs des capacités maximum sont entières, $G_f(1) = G_f$. Par le lemme 9.19, f est alors un flot maximum. Un point subtil est que ce raisonnement repose sur le fait qu'il *existe* un dernier tour de boucle : c'est pour cela qu'on demande $k \geq 1$ lors de l'initialisation.

Lors de l'étape d'amélioration du float à l'étape 2(a), la valeur du flot f est augmentée d'au moins 2^{k-1}, puisque tous les arcs de $G_f(2^{k-1})$ ont un coût d'au moins 2^{k-1}, par définition. Sa valeur au début de la boucle de l'étape 2(a) est d'au moins $v_{\max} - m2^k$ par **(Inv 1)**, et ne peut pas dépasser $v_{\max}$, par définition de $v_{\max}$. Donc la boucle de l'étape 2(a) effectue au maximum $2m$ tours.

La complexité de la recherche d'un chemin améliorant est un simple calcul d'accessibilité dans $G_f(2^{k-1})$ (voir les détails plus bas), qui se fait en temps $O(m)$, de même que la mise à jour du flot. L'étape 2(a) prend donc un temps $O(m^2)$; ou plutôt, un temps majoré par une fonction affine en m^2, avec des coefficients constants.

Comme k décroît de 1 à chaque tour de la boucle externe (étape 2), et est initialisé à un nombre de l'ordre de $\log_2 C$, et supérieur ou égal à 1, la complexité de l'étape 2 est donc en $O(m^2 \max(1, \log C))$. On doit y ajouter le temps de l'initialisation, dont on a vu qu'il était en $O(m)$. □

L'étape de recherche de chemin améliorant est beaucoup plus simple que dans l'algorithme à distances estimées de Dinic-Edmonds-Karp. Il suffit de construire le graphe $G_f(D)$ (pour $D \stackrel{\text{def}}{=} 2^{k-1}$), et de faire un test d'accessibilité en profondeur comme par `reach_set` (voir la table 3), qui maintienne en plus dans une variable $\mathtt{p_T}$ la fonction prédécesseur pT de l'arbre couvrant $T \stackrel{\text{def}}{=} (S, s, p_T)$ associé au parcours (voir la proposition 4.5 à la section 5.4). Le code, `augmenting-path`, en est donné à la table 27, où l'on a aussi modifié le code de sorte à travailler directement sur le graphe G, sans construire explicitement $G_f(D)$. Le tableau `pT` est mis à jour aux lignes 10 et 14. Ce tableau n'a pas à être initialisé (ses valeurs seront de toute façon réécrites au cours de l'exécution de l'algorithme) sauf pour $\mathtt{pT}[p]$, que nous initialiserons à une valeur indéfinie `undef` avant d'appeler `augmenting-path` : en sortie de `augmenting-path`, il y aura un chemin de s à p dans $G_f(D)$, donc un chemin augmentant, si et seulement si $\mathtt{pT}[p] \neq \mathtt{undef}$. On maintient aussi un tableau de booléens `conforme` qui dit si l'arc $u \to_T v$ (i.e., $u = \mathtt{pT}[v]$) est conforme ou pas. (Voir la remarque 9.11, et le tableau de même nom utilisé dans l'algorithme de Dinic-Edmonds-Karp à la table 25.)

La complexité de `augmenting-path` en majorée par quelque chose de proportionnel à $\sum_{x \in S}(\#\mathtt{succ}(x) + \#\mathtt{pred}(x) + 1) = O(n + m)$, qui est donc en $O(m)$ par la remarque 9.38. On n'a pas besoin de construire de chemin augmentant

```
   fun augmenting-path (G as (S, succ, pred), s, f, cmax, D) =
1      work := empty; for each u ∈ S do push (u, work);
2      marked := {s};
3      while nonempty(work)) do
4            let u=pop(work) in (* on dépile le prochain sommet à traiter u *)
5                if u ∉ marked (* si u n'est pas déjà marqué, *)
7                then (for each v ∈ succ(u) do (* explorer ses successeurs
                                                  par arcs conformes, de coûts ≥D *)
8                                  if f(u,v)<cmax(u,v) and cmax(u,v)-f(u,v)>=D
9                                     then (push (v, work);
10                                          pT[v] := u; conforme[v] := true);
11                     for each v ∈ pred(u) do (* explorer ses successeurs
                                                  par arcs non conformes, de coûts ≥D *)
12                                 if f(v,u)>0 and f(v,u)>=D
13                                    then (push (v, work);
14                                          pT[v] := u; conforme[v] := false);
15                     marked:=marked ∪ {u}); (* et marquer u *)
```

TABLE 27 – La recherche de chemin améliorant dans $G_f(D)$

explicitement, sous forme de liste par exemple, et il suffira de le lire à rebours, en partant du puits p, en suivant les pointeurs donnés dans le tableau `pT` jusqu'à atteindre s.

Pour mettre à jour le flot, on peut appeler la fonction `update-flow` de la table 23, mais pour ceci il faut d'abord convertir la représentation du chemin améliorant via tableau de prédécesseurs `pT` est tableau de booléens `conforme` (indexé par les sommets) en une représentation par tableau de sommets `mu` (jusqu'à l'indice `nmu`) et tableau `conforme` (indexé par les indices de sommets dans le tableau `mu`). On peut aussi directement le calculer par la procédure de la table 29, qui a la même complexité, en $O(n)$.

L'algorithme par échelonnement d'Edmonds-Karp et Dinic est affiché à la table 32. La procédure `fitting-bit-mask` de la table 31 est une implémentation en temps constant de la recherche du plus petit entier $k \geq 1$ tel que $2^k \geq n$; il est fondé sur l'astuce qu'en complément à 2, pour tout entier n strictement positif, $n\&(-n)$ est égal à la plus grande puissance de deux inférieure ou égale à n. L'opération $k\ll m$ calcule k décalé de m bits à gauche ; donc `k«1` calcule simplement le double de `k`. Cette opération, plutôt que de retourner k, retourne $D \stackrel{\text{def}}{=} 2^k$. De la sorte, au lieu de décrémenter k, on divises D par deux, en utilisant l'opération de décalage à droite `»` à la ligne 9 de la table 32.

Remarque 9.45 *Dans le cas d'entiers machine, il faut faire attention aux débordements arithmétiques (`-n`, `k«1`) ; dans le cas d'entiers en grande précision,*

```
   fun alpha-pT (f, cmax, pT, conforme, s, p) =
1      u := pT[p];
2      a := rf (f, cmax, u, p, conforme[p]);
3      while u≠s do
4            let new-a = rf (f, cmax, pT[u], u, conforme[u]) in
5                (u := pT[u];
6                 if new-a<a
7                    then a := new-a); (* calcul du minimum *)
8      return a;
```

TABLE 28 – Le calcul de $\alpha(\mu)$ où μ est un chemin améliorant donné par prédécesseurs `pT` dans un arbre couvrant du graphe d'écart

```
   fun update-flow-pT (f, cmax, pT, conforme, s, p) =
1      a := alpha-pT (f, cmax, pT, conforme, s, p);
2      u := p;
3      while u≠s do
3            (if conforme[u]
4                then f (pT[u], u) += alpha; (* arc conforme *)
5             else f (u, pT[u]) -= alpha; (* arc non conforme;
                                             attention au renversement de l'arc! *)
6             u := pT[u])
```

TABLE 29 – La mise à jour d'un flot par un chemin améliorant donné par prédécesseurs `pT` dans un arbre couvrant du graphe d'écart

```
   fun fitting-bit-mask (n) =
1      if n=0 then return 1;
2      let a = n & (-n) in
3          if a=n
4             then return a;
5          else return a«1;
```

TABLE 30 – Le calcul de 2^k où k est le plus petit entier $k \geq 0$ tel que $2^k \geq$ `n` (les entiers sont supposés codés en complément à 2)

```
   fun init-bit-mask (S, succ, cmax) =
1      D := 2; (* On souhaite k ≥ 1, donc 2^k ≥ 2 *)
2      for each x ∈ S do
3               for each y ∈ succ(x) do
4                        D := max (D, fitting-bit-mask (cmax (x, y)));
5      return D;
```

TABLE 31 – Le calcul de 2^k où k est le plus petit entier $k \geq 1$ tel que $2^k \geq$ `n` (les entiers sont supposés codés en complément à 2)

```
   fun echelonnement (G as (S, succ, pred), s, p, cmax) =
1      for each x ∈ S do
2               for each y ∈ succ(x) do
3                        f(x,y) := 0; (* initialisation du flot *)
4      D = init-bit-mask (S, succ, cmax);
5      while D≥2 do
6            (while (pT[p] := undef, augmenting-path (G, s, f, cmax, D»1),
7                        pT[p]≠undef) do
                         (* on a trouvé un chemin améliorant *)
8                  update-flow-pT (f, cmax, pT, conforme, s, p);
9             D := D»1);
10     return f;
```

TABLE 32 – L'algorithme par échelonnement d'Edmonds-Karp et Dinic

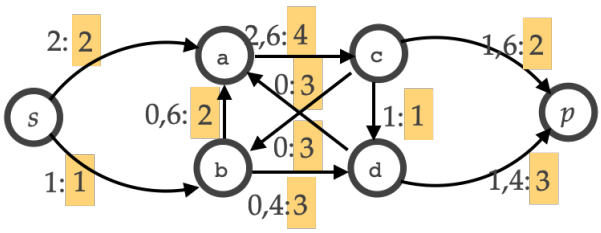


FIGURE 8 – Un flot maximum à valeurs non entières sur un réseau dont les capacités maximums sont toutes entières

on notera que cette procédure est en temps proportionnelle à la taille des entiers, c'est-à-dire à k, qui est majoré par $\log_2 C$. Dans ce modèle d'entiers en grande précision, où les entiers sont représentés comme des listes d'entiers machines, la taille des entiers calculés au cours de l'algorithme par échelonnement d'Edmonds-Karp et Dinic est en $O(\log_2(nC))$ (la somme des valeurs d'un flot sur un chemin améliorant, de longueur au plus n), et la complexité de l'algorithme est en fait en $O(m^2 \log C(\log C + \log n))$.

9.9 Chemins disjoints, et le théorème de Menger

Théorème 9.46 *Pour tout réseau (G, s, p), si les capacités maximums des arcs sont toutes entières, alors il en est de même de la valeur du flot maximum, et il existe un flot maximum f tel que $f(e)$ soit entier pour tout arc e de G.*

On a le même résultat en remplaçant « entier » par « multiple entier d'une valeur $c > 0$ fixée » ou par « rationnel ».

Démonstration. À toute étape du calcul dans l'algorithme de Ford-Fulkerson (et donc dans toutes ses variantes dont on a montré la terminaison, l'algorithme de Dinic-Edmonds-Karp ou l'algorithme par échelonnement d'Edmonds-Karp et Dinic), le flot f est à valeurs entières, c'est-à-dire que $f(e)$ est un entier pour tout arc e ; alors les valeurs de $\alpha(\mu)$, pour n'importe quel chemin améliorant sont entières, ce qui réinstalle l'invariant après une étape d'amélioration du flot.

Le cas où toutes les capacités maximums sont multiples entières d'une constant $c > 0$ est similaire, ou bien conséquence du cas entier à condition de diviser toutes les capacités maximales par c, puis de multiplier le flot maximal à valeurs entières ainsi obtenu par c. Le cas des capacités rationnelles en est un cas particulier : toutes les capacités sont multiples de $1/N$, où N est le ppcm des dénominateurs des capacités des arcs de G. □

Remarque 9.47 *Supposons que les capacités maximums des arcs soient entières. Le théorème 9.46 ne dit pas que tout flot maximum est à valeur entières, juste qu'il en existe un qui est à valeurs entières. La figure 8 montre un réseau à capacités*

maxmums entières (sur fond orange), et un flot maximum (les chiffres précédant les capacités maximums) dont les valeurs ne sont pas entières.

Le théorème 9.46 s'applique en particulier lorsque les capacités maximums sont toutes égales à 1. Aussi trivial que ce cas particulier puisse sembler, il est important.

Considérons notamment le problème suivant.

chemins disjoints

Définition 9.48 *Le problème des* chemins disjoints *est le suivant :*
Entrée *: un graphe orienté G, deux sommets distincts s et p de G, et un entier k écrit en binaire ;*
Question *: G a-t-il au moins k chemins deux à deux disjoints, c'est-à-dire n'ayant aucun arc en commun, de s à p ?*

Lorsque $k = 1$, il s'agit simplement du problème d'accessibilité dans les graphes orientés. Pour $k \geq 2$, le problème est plus complexe. Si l'on cherche d'abord un premier chemin de s à p, puis un deuxième, il faudra possiblement itérer sur le choix du premier chemin jusqu'à trouver un deuxième chemin disjoint du premier. Cette stratégie algorithmique ne semble pas devoir mener à un algorithme plus efficace qu'exponentiel, car il risque de devoir énumérer tous les chemins élémentaires de s à p. La théorie des flots maximums nous fournit un algorithme en temps polynomial, cependant, grâce au lemme suivant.

Lemme 9.49 *Soit G un graphe orienté, et s et p deux sommets distincts de G. Posons G' le sous-graphe induit de G dont les sommets sont les sommets qui sont à la fois accessibles depuis s et coaccessibles depuis p dans G. On définit la capacité maximum de tout arc de G' comme valant 1. Pour tout $k \in \mathbb{N}$, il y a au moins k chemins deux à deux disjoints de s à p dans G si et seulement si la valeur du flot maximum du réseau (G', s, p) est supérieure ou égale à k.*

Démonstration. S'il y a k chemins deux à deux disjoints de s à p dans G, d'abord on peut les supposer élémentaires, comme pour le lemme de Kőnig (lemme 4.3). Ensuite, tous les sommets de ceux chemins sont accessibles depuis s et coaccessibles depuis p dans G, donc ces k chemins sont aussi des chemins de G'. La fonction f associant la valeur 1 à tout arc présent sur un de ces chemins élémentaires respecte la condition d'admissibilité de la définition 9.2. Elle respecte aussi la condition de conservation du flot, car en tout sommet $x \neq s, p$, chaque chemin contribue exactement un arc entrant en x et un arc sortant en x, et ils sont tous distincts deux à deux. Donc f est un flot. On constate maintenant que $\mathrm{out}(f, \{s\})$ vaut exactement k, donc $\mathrm{val}(f) \geq k$.

Réciproquement, on démontre que si (G', s, p) a un flot f, que l'on peut supposer à valeurs entières, et tel que $\mathrm{val}(f) \geq k$, alors il existe k chemins deux à deux disjoints de s à p dans G' — et donc dans G — par récurrence sur $k \in \mathbb{N}$; nous montrerons de plus que les arcs e des k chemins peuvent être choisis de

sorte à ce que $f(e) = 1$. (Nous appellerons plus brièvement ces arcs des *f-arcs.*) Si $k = 0$, c'est évident. Supposons donc $k \geq 1$.

Soit G'' le sous-graphe induit de G' composé des f-arcs, et E l'ensemble des sommets accessibles depuis s dans G''. Si $p \notin E$, E est une coupe de (G', s, p). La valeur $\text{out}(f, E)$ est la somme sur les arcs sortants $u \to v$ des quantités $f(u, v)$, qui sont par définition toutes nulles. Donc $\text{val}(f) = \text{out}(f, E) - \text{in}(f, E) \leq \text{out}(f, E) = 0$, ce qui contredit le fait que $\text{val}(f) \geq k \geq 1$. Donc $p \in E$, ce qui signifie qu'il existe un chemin γ de s à p dans G'', que l'on peut supposer élémentaire par le lemme de Kőnig (lemme 4.3).

Soit f' le flot obtenu à partir de f en mettant à 0 les valeurs des arcs sur le chemin élémentaire γ. Une façon simple de voir que f' est un flot est de réaliser que $f' = f.\mu$, où μ est le chemin formé des arcs de γ pris à l'envers, et donc vus comme des arcs non conformes dans le graphe d'écart G'_f, et d'appliquer le lemme 9.17. On a $\text{out}(f', \{s\}) = \text{out}(f, \{s\}) - 1$, puisqu'exactement un arc sortant de s est concerné par la mise à jour. On a aussi $\text{in}(f', \{s\}) = \text{in}(f, \{s\})$ puisqu'aucun arc entrant sur s ne peut faire partie de γ, γ étant élémentaire. Donc $\text{val}(f') = \text{val}(f) - 1 \geq k - 1$. Par hypothèse de récurrence on peut trouver au moins $k-1$ chemins deux à deux disjoints formées de f'-arcs, et comme aucun arc de γ n'est un f'-arc par définition, ces $k-1$ chemins sont tous disjoints de γ. □

Théorème 9.50 *Le problème des chemins disjoints est décidable en temps $O(n+m^2)$.*

Démonstration. On construit d'abord le graphe G' du lemme 9.49. Il suffit d'appeler la fonction `reach_set` sur G et s et sur G renversé et p pour obtenir l'ensemble E des sommets accessibles depuis s et l'ensemble F des sommets coaccessibles depuis p, en temps $O(n + m)$. L'opération d'intersection des deux ensembles se fait en temps $O(n)$ en représentation des marques par tableaux de booléens (voir la section 4.5), si on souhaite l'effectuer (voir plus bas).

Le point important est qu'il ne reste plus qu'à appeler l'algorithme par échelonnement d'Edmonds-Karp et Dinic. Dans ce cas, $C = 1$, donc $\max(1, \log C) = 1$, et sa complexité est en $O(m^2)$ par le théorème 9.44. □

Remarque 9.51 *La construction de G' au début de la démonstration du théorème 9.50 n'est pas nécessaire. En effet, et même si (G, s, p) n'est pas nécessairement un réseau (certains sommets étant possiblement inaccessibles depuis s ou coinaccessibles depuis p), l'algorithme par échelonnement d'Edmonds-Karp et Dinic appliqué à (G, s, p) n'explore de toute façon que des chemins passant par des sommets de G', donc dans G' puisque G' est un sous-graphe induit. Mais alors l'initialisation s'effectue en temps $O(n+m)$, et la recherche de chemin améliorant aussi, car on ne peut plus profiter de la remarque 9.38. La complexité est alors en $O(m(n + m)\max(1, \log C))$, est comme $C = 1$, en $O(m(n + m))$, ce qui est moins bon que la complexité donnée au théorème 9.50.*

```
   fun disjoint-paths(G as (S, succ, pred), s, p) =
1      let reach = reach_set ((S, succ), s) in
2      let coreach = reach_set ((S, pred), p) in
3      let fun ok(u) = u ∈ reach and v ∈ coreach in
4      let S' = [u | u ∈ S such that ok(u)] in
5      let memofun succ'(u) = [v | v ∈ succ(u) such that ok(v)] in
6      let memofun pred'(u) = [v | v ∈ pred(u) such that ok(v)] in
7      let trivial-cmax(u,v) = 1 in
8      let f = echelonnement ((S', succ', pred'), s, p, trivial-cmax) in
9          return flow-val (f, G, s)
```

TABLE 33 – L'algorithme de calcul du nombre maximal de chemins disjoints de s à p ($s \leq p$) dans un graphe G

Remarque 9.52 *Le terme en $+n$ dans la complexité énoncée au théorème 9.50 est souvent oublié. Il n'est pas nécessaire si $n \leq m^2$, ce qui est souvent le cas.*

Une réalisation possible de cet algorithme, qui calcule directement le nombre maximum de chemins disjoints de s à p. On y utilise quelques facilités d'écriture importées d'autres langages. La notation `[u | ...]` est une compréhension de liste, inspirée de Haskell et de GimML. Par exemple, à la ligne 4, la partie `u ∈ S` entre `|` et `such that` introduit une variable liée `u` qui doit parcourir la liste `S`, et seules les valeurs de `u` satisfaisant au prédicat après le mot-clé `such that` seront listées. Le mot-clé `memofun` implémente une fonction *mémoïsante* : chacune de ces fonctions F (ici, `succ'` à la ligne 5 ou de `pred'` à la ligne 6) est associée à une table T_F, créée au moment de la déclaration de la fonction, et qui contient tous les résultats des calculs effectués sur des arguments précédentes. Autrement dit, une déclaration de la forme :

`memofun` F`(u) = ⟨e⟩`

est une abréviation pour le code suivant, modulo quelques notations non définies (`new-table()`, `u` $\in T_F$, T_F`[u]`, T_F`[u] := res`, ou `fun u =>` qui introduit une fonction anonyme d'argument formel `u`), que j'espère compréhensibles, et dont vous verrez comment les implémenter vous-même :

```
let F= (let T_F= new-table() in
            fn u => if u ∈ T_F
                       then return T_F[u]
                    else let res = ⟨e⟩ in
                             (T_F[u] := res;
                              return res))
```

Finalement, le calcul de la valeur du flot est effectué à la ligne 9 par un appel à la fonction `flow-val` de la table 26.

Nous terminons cette section sur une application directe due à Menger (Karl Menger, *Zur allgemeinen Kurventheorie*, Fund. Math. 10 :96–115, 1927).

Théorème 9.53 (Menger) *Soit G un graphe, et s et p deux sommets disjoints de G. Le nombre maximum de chemins disjoints de s à p dans G est égal au nombre minimum d'arcs de G dont la suppression rend p inaccessible depuis s.*

Démonstration. Le nombre maximum k de chemins disjoints de s à p dans G est égal au flot maximum du réseau (G', s, p) défini au lemme 9.49. Par le théorème max-flow min-cut (théorème 9.32), c'est aussi la valeur minimum d'une coupe E de (G', s, p).

Comme toutes les capacités maximums valent 1, la valeur k de la coupe E est exactement le nombre d'arcs sortants de E. Si l'on supprime ces k arcs sortants de G, alors on obtient un graphe G'' dans lequel p n'est plus accessible depuis s : sinon un tel chemin serait dans G', et l'un des arcs de ce chemin doit sortir de E, puisque par définition d'une coupe $s \in E$ mais p n'est pas dans E. Ceci montrer que le nombre minimum d'arcs de G à supprimer pour rendre p inaccessible depuis s est inférieur ou égal à k.

Réciproquement, si l'on supprime strictement moins de k arcs de G, alors l'un des k chemins disjoints de s à p dans G n'a aucun arc supprimé, donc le graphe obtenu en supprimant ces arcs est toujours tel que p y est accessible depuis s. □

9.10 Couplages dans les graphes bipartis

Nous nous intéressons brièvement aux graphes *non orientés*.

Définition 9.54 *Un* couplage *(« matching » en anglais) dans un graphe non orienté G est un ensemble d'arêtes sans sommet commun.* couplage

Nous nous intéressons plus particulièrement ici aux graphes bipartis, définis comme suit.

Définition 9.55 *Un* graphe biparti *est un triplet (S_1, S_2, A), où S_1 et S_2 sont deux ensembles finis disjoints, et $A \subseteq S_1 \times S_2$.* graphe biparti

Un graphe biparti (S_1, S_2, A) définit un graphe non orienté $(S_1 \cup S_2, \{\{x, y\} \mid (x, y) \in A\}$. On pourrait aussi le voir comme un graphe orienté, ce que nous ferons d'ailleurs au lemme 9.57 ci-dessous.

Remarque 9.56 *Un graphe biparti est souvent défini comme un graphe non orienté $G \stackrel{def}{=} (S, A)$ est* biparti *si et seulement son ensemble de sommets S s'écrit comme l'union disjointe $S_1 \uplus S_2$ de deux ensembles, de sorte que toute arête u — v*

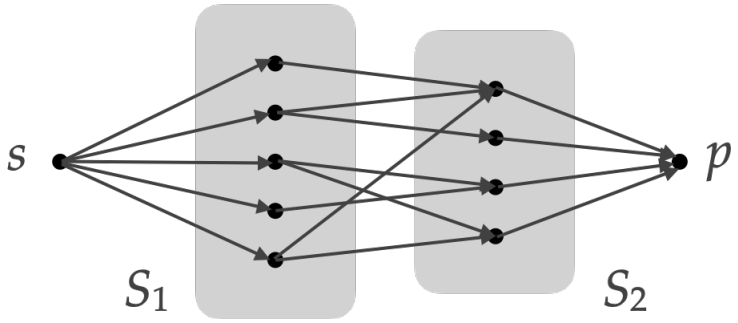


FIGURE 9 – La réduction du problème des couplages dans un graphe biparti (sommets sur fond gris) à un problème de flot maximum

relie un sommet de S_1 à un sommet de S_2. Ce n'est pas une définition équivalente à la définition 9.55, car elle n'inclut pas la spécification de quels ensemble S_1 et S_2 choisir, contrairement à la définition 9.55. Disons que cette nouvelle définition est celle d'un graphe non orienté implicitement biparti*. A partir d'un graphe non orienté implicitement biparti, on peut retrouver S_1 et S_2 satisfaisant au conditions ci-dessus (et même en temps $O(n+m)$), mais pas de façon unique. Cette non-unicité ne cause aucun problème dans les développements qui suivent.*

Trouver un couplage de cardinalité maximum dans un graphe biparti est facile, car ceci se ramène à un problème de flot maximum. Le réseau (G_0, s, p) construit ci-dessous est illustré à la figure 9, le graphe biparti d'origine est le sous-graphe induit de G_0 par les sommets sur fond gris.

Lemme 9.57 *Soit $G \stackrel{def}{=} (S_1, S_2, A)$ un graphe biparti. Notons G_0 le réseau obtenu en ajoutant à G deux sommets distincts s et p, plus :*

- *des arcs $s \to u$ pour chaque sommet $u \in S_1$, de capacité maximum 1 ;*
- *des arcs $u \to v$ pour chaque couple $(u, v) \in A$, de capacité maximum 1 ;*
- *des arcs $v \to p$ pour chaque sommet $v \in S_2$, de capacité maximum 1.*

La fonction φ qui à chaque couplage C de G associe l'unique flot f tel que $f(u, v) = 1$ pour tout arc (u, v) de C, $f(u, v) = 0$ pour tout arc $(u, v) \in A \setminus C$ est une bijection de l'ensemble des couplages de G vers l'ensemble des flots à valeurs entières de (G_0, s, p).

Un flot f est *à valeurs entières* si et seulement si $f(e)$ est entier pour tout arc e. On prendra soin de ne pas confondre cette notion avec un flot *à valeur entière*, qui est un flot f tel que $\text{val}(f)$ est entier.

à valeurs entières

à valeur entière

Démonstration. D'abord, $\varphi(C)$ est bien défini. Ceci revient à dire qu'il existe bien un unique flot $f \stackrel{\text{def}}{=} \varphi(C)$ satisfaisant les conditions données plus haut. Il suffit pour cela de vérifier que $f(s, u)$ et $f(v, p)$ est bien défini pour tout $u \in S_1$ et pour tout $v \in S_2$. L'équation de conservation du flot nous donne en effet que

$f(s,u)$ est égal au nombre d'arcs d'origine u qui sont dans le couplage C, et que $f(v,p)$ est égal au nombre d'arcs d'extrémité v qui sont dans C. Vérifions la condition d'admissibilité. Pour tout sommet $u \in S_1$, il existe au plus un arc $u \to v$ d'origine u dans C, donc tel que $f(u,v) = 1$; les autres ont pour valeur 0. Donc $f(s,u)$ vaut 1 s'il existe un arc d'origine u dans C, et 0 sinon. Dans les deux cas, la valeur est inférieure à la capacité maximum de l'arc $s \to u$, qui vaut 1. On raisonne de façon similaire pour les sommets v de S_2 et les capacités maximum des arcs $v \to p$.

Réciproquement, soit f un flot à valeurs entières sur (G_0, s, p). On vérifie que l'ensemble $\psi(f) \stackrel{\text{def}}{=} \{(u,v) \in A \mid f(u,v) = 1\}$ est un couplage comme suit. Soit u un sommet quelconque de S_1. Comme la capacité maximum de l'arc $s \to u$ vaut 1, par l'équation de conservation du flot en u, au plus un arc d'origine u est dans $\psi(f)$. Similairement, en raisonnement sur l'arc $v \to p$, tout sommet $v \in S_2$ est extrémité d'au plus un arc de $\psi(f)$.

Pour tout couplage C, $\psi(\varphi(C))$ est l'ensemble des arêtes $(u,v) \in A$ tel que $\varphi(C)(u,v) = 1$, c'est-à-dire tel que $(u,v) \in C$. Pour tout flot à valeurs entières f sur (G_0, s, p), $\varphi(\psi(f))$ est l'unique flot f' tel que $f'(u,v) = 1$ pour tout $(u,v) \in C$ (c'est-à-dire pour tout arc $(u,v) \in A$ tel que $f(u,v) = 1$), et $f'(u,v) = 0$ pour tout $(u,v) \in A \setminus C$ (c'est-à-dire pour tout arc $(u,v) \in A$ tel que $f(u,v) \neq 1$). Or, comme f est à valeurs entières, et que $0 \leq f(e) \leq c_{\max}(e) \stackrel{\text{def}}{=} 1$ pour tout arc e, $f(u,v) \neq 1$ est équivalent à $f(u,v) = 0$. Donc $f' = f$, ce qui montrer que les deux fonctions φ et ψ sont inverses l'une de l'autre. □

La construction de (G_0, s, p) à partir de G s'effectue en temps $O(n)$. En appliquant l'algorithme par échelonnement d'Edmonds-Karp et Dinic (avec $C = 1$), on obtient donc le résultat suivant.

Théorème 9.58 *Étant donné un graphe biparti G, on peut trouver un couplage de cardinalité maximum sur G en temps $O(n + m^2)$.*

Comme précédemment, on trouve souvent dans la littérature une affirmation que la complexité est en $O(m^2)$; voir la remarque 9.52. Le résultat est initialement dû à Kuhn (H. W. Kuhn, *The Hungarian Method for the Assignment Problem*, Naval Research Logistic Quarterly 2 :83–97, 1955), et Munkres (J. Munkres, *Algorithms for the Assignment and Transportation Problems*, J. Soc. Ind. Appl. Math. 5(1) :32–38, 1957), qui en a le premier démontré la complexité polynomiale. Cette complexité avait été évaluée à $O(n^4)$, car Kuhn et Munkres représentaient leurs graphes sous forme de matrices d'adjacence, pas par listes de successeur.

L'instantiation de l'algorithme par échelonnement à ce cas particulier est appelé l'*algorithme hongrois*, en référence à D. Kőnig et J. Egerváry, dont Kuhn dit s'être inspiré. Selon Martello (voir l'article Wikipedia « algorithme hongrois », et la référence : S. Martello, *Jenö Egerváry : from the origins of the Hungarian algorithm to satellite communication*, Central European J. Oper. Res 18(1) :47–58,

2010), il a été découvert en 2006 que Jacobi avait décrit l'algorithme au XIXème siècle, dans un article posthume.

Le problème du calcul d'un couplage de cardinalité maximum dans un graphe *non nécessairement biparti* est beaucoup plus compliqué, mais se fait toujours en temps polynomial. C'est un algorithme célèbre, dû à Edmonds (J. Edmonds, *Paths, Trees, and Flowers*, Can. J. Math. 17 :449-467, 1965). Une fois compris le cas biparti, la difficulté vient du traitement des cycles de longueurs impaires. Un exposé des idées principales se trouve dans des notes de cours de Tarjan (R. E. Tarjan, *Sketchy Notes on Edmonds' Incredible Shrinking Blossom Algorithm for General Matching*, cours COS 423 : Theory of Algorithms, printemps 2022, disponible en `https://www.cs.dartmouth.edu/~ac/Teach/CS105-Winter05/Handouts/tarjan-blossom.pdf`).

9.11 Algorithmes de préflots

On peut produire des algorithmes de calcul de flots maximum plus efficace que les algorithmes précédentes à l'aide la notion de préflot.

Définition 9.59 *Un* préflot *sur un réseau (G, s, p) est une fonction $f : A \to \mathbb{R}_+$,* préflot
où A est l'ensemble des arcs de G, telle que :

1. *Pour tout sommet u de G différent de s, l'*excès *de f en u, défini comme* excès
   $\mathrm{e}_f(u) \stackrel{def}{=} \sum_{v/v\to u} f(v,u) - \sum_{w/u\to w} f(u,w)$*, soit positif ou nul ;*
2. *pout tout arc e, $f(e) \leq c_{\max}(e)$.*

Un préflot est donc un flot si et seulement si l'excès de f en tout sommet autre que s ou p est nul.

L'algorithme que nous allons développer est dû à Goldberg et Tarjan (A. V. Goldberg, R. E. Tarjan, *A New Approach to the Maximum Flow Problem*, Proc. 18th ACM Symp. Th. Comp. (STOC), pages 136–146, 1986), et avant eux à Karzanov (A. V. Karzanov, *Nakhozhdenie maksimal'nogo potoka v seti metodom predpotokov*, Doklady Akad. Nauk SSSR, 215(1) :49–52, 1974, traduction en anglais : *Determining the Maximum Flow in a Network by the Method of Preflows*, Soviet Math. Dokl. 15 :434–437, 1974).

L'algorithme fonctionne par étapes de *poussée* de l'excès du préflot courant de s vers p, et de *réétiquetage*, c'est-à-dire de mise à jour d'une distance estimée, comme à la section sec :lalgorithme-de-dinic. Les distances estimées sont définies à la section 9.5.

Il est très facile de trouver un préflot, et le préflot initial sera le suivant.

Lemme 9.60 *Soit (G, s, p) un réseau, et $c_{\max}$ sa fonction de capacité maximum. La fonction f_0 qui à tout arc e d'origine s associe $c_{\max}(e)$ et à tout autre arc associe 0 est un préflot. On l'appelle le* préflot initial. préflot initial

Démonstration. La condition d'admissibilité est claire. Pour tout sommet $u \neq s$, les arcs $u \to w$ d'origine u ont une valeur de flot $f_0(u \to w)$ égale à 0, donc $\mathrm{e}_{f_0}(u) = \sum_{v/v\to u} f_0(v \to u) \geq 0$. □

Définition 9.61 *Soit f un préflot sur un réseau (G, s, p). Un sommet u de G est* actif *pour f si et seulement si $u \neq s$, $u \neq p$ et $\mathrm{e}_f(u) > 0$.* actif

Étant donné un sommet actif u est un arc $a\colon u \to_f v$ admissible (c'est-à-dire tel que $\Delta(u) = \Delta(v) + 1$, voir la définition 9.23), l'opération de poussée $f := f.a$ définie ci-dessous consiste à transférer le plus possible de l'excès $\mathrm{e}_f(u)$ le long de l'arc de G dont l'arg a est le représentant. On ne peut pas transférer plus que $r_f(a)$, est l'on transférera donc $\min(\mathrm{e}_f(u), r_f(a))$.

Définition 9.62 (Poussée) *Pour tout préflot f sur un réseau (G, s, p), pour tout arc $a\colon u \to_f v$ du graphe d'écart donc l'origine est un sommet actif u pour f, on définit la fonction $f.a$ des arcs de G vers $\mathbb{R}_+$ par :*

- *si a est représentant conforme d'un arc $u \to v$ de G, alors $(f.a)(u, v) \stackrel{def}{=} f(u, v)+\epsilon$, où $\epsilon \stackrel{def}{=} \min(\mathrm{e}_f(u), c_{\max}(u, v)-f(u, v))$ (autrement dit, $(f.a)(u, v) \stackrel{def}{=} \min(\mathrm{e}_f(u) + f(u, v), c_{\max}(u, v))$), et $(f.a)(e) \stackrel{def}{=} f(e)$ pour tout autre arc e de G ;*
- *si a est représentant non conforme d'un arc $v \to u$ de G, alors $(f.a)(v, u) \stackrel{def}{=} f(v, u)-\epsilon$, où $\epsilon \stackrel{def}{=} \min(\mathrm{e}_f(u), f(v, u))$ (autrement dit, $(f.a)(v, u) = \max(f(v, u)-\mathrm{e}_f(u), 0)$), et $(f.a)(e) \stackrel{def}{=} f(e)$ pour tout autre arc e de G.*

Lemme 9.63 *Sous les hypothèses de la définition 9.62, $f.a$ est un préflot.*

Démonstration. L'excès de $f.a$ en tout sommet différent de u et de v est le même que celui de f, car $f.a$ ne diffère de f qu'en un unique arc $u \to v$ ou $v \to u$.

Si a est représentant conforme de $u \to v$:

- L'excès $\mathrm{e}_{f.a}(u)$ de $f.a$ en u vaut $\sum_{x/x\to u}(f.a)(x, u) - \sum_{y/u\to y}(f.a)(u, y) = \sum_{x/x\to u} f(x, u)-\sum_{y\neq v/u\to y} f(u, y)-f(u, v)-\epsilon = \mathrm{e}_f(u)-\epsilon$, où $\epsilon = \min(\mathrm{e}_f(u), c_{\max}(u, v)-f(u, v))$. Donc $\mathrm{e}_{f.a}(u) = \max(0, \mathrm{e}_f(u) - c_{\max}(u, v) + f(u, v)) \geq 0$.
- L'excès $\mathrm{e}_{f.a}(v)$ de $f.a$ en v vaut $\sum_{x/x\to v}(f.a)(x, v) - \sum_{y/v\to y}(f.a)(u, y) = \sum_{x\neq u/x\to v} f(x, v) + f(u, v) + \epsilon - \sum_{y/v\to y} f(v, y) = \mathrm{e}_f(v) + \epsilon \geq 0$.

Si a est représentant non conforme de $v \to u$:

- L'excès $\mathrm{e}_{f.a}(u)$ de $f.a$ en u vaut $\sum_{x/x\to u}(f.a)(x, u) - \sum_{y/u\to y}(f.a)(u, y) = \sum_{x\neq v/x\to u} f(x, u)+f(v, u)+\epsilon-\sum_{y/u\to y} f(u, y) = \mathrm{e}_f(u)+\epsilon$, où $\epsilon = \min(\mathrm{e}_f(u), f(v, u))$. Ceci est bien positif ou nul.
- L'excès $\mathrm{e}_{f.a}(v)$ de $f.a$ en v vaut $\sum_{x/x\to v}(f.a)(x, v) - \sum_{y/v\to y}(f.a)(u, y) = \sum_{x/x\to v} f(x, v) - \sum_{y\neq u/v\to y} f(v, y) - f(v, u) - \epsilon$ (avec ϵ comme ci-dessus) $= \mathrm{e}_f(v) - \epsilon = \max(0, \mathrm{e}_f(v) - f(v, u)) \geq 0$. □

Comme précédemment, nous allons définir un système de transition. Les configurations en sont des couples (f, Δ) où f est un préflot et Δ est une distance estimée relative à f.

Définition 9.64 *Soit (G, s, p) un réseau. Une* transition de poussée *est de la forme $(f, \Delta) \rightsquigarrow (f.a, \Delta)$, où f est un flot sur (G, s, p), Δ est une distance estimée relative à f, a est un arc admissible du graphe d'écart, d'origine un sommet actif u.*

transition de poussée

Elle est saturante *si et seulement si* $\mathrm{e}_f(u) \geq r_f(a)$.

saturante

Remarque 9.65 *Dans les conditions de la définition 9.64, on ne retrouve pas l'arc $u \to_{f.a} v$ dans le graphe d'écart $G_{f.a}$ si et seulement si la poussée est saturante et $r_f(a) > 0$. En effet, si a est représentant conforme de $u \to v$, alors il disparaît de $G_{f.a}$ si et seulement si $(f.a)(u, v) = c_{\max}(u, v)$ (et $f(u, v) < c_{\max}(u, v)$, donc $r_f(a) > 0$ puisque a est dans G_f), si et seulement si $\mathrm{e}_f(u) + f(u, v) \geq c_{\max}(u, v)$, si et seulement si $\mathrm{e}_f(u) \geq r_f(a)$; et si a est représentant non conforme de $v \to u$, alors il disparaît de $G_{f.a}$ si et seulement si $(f.a)(v, u) = 0$ (et $f(v, u) = r_f(a) > 0$, car a est dans G_f), si et seulement si $f(v, u) \leq \mathrm{e}_f(u)$, si et seulement si $\mathrm{e}_f(u) \geq r_f(a)$.*

Par le lemme 9.63, $f.a$ est encore un préflot sur (G, s, p). On vérifie, comme au lemme 9.26, que Δ reste une distance estimée relative à $f.a$.

Lemme 9.66 *Soit f un préflot sur un réseau (G, s, p), et Δ une distance estimée relative à f. Pour tout arc admissible $a \colon u \to_f v$ du graphe d'écart, d'origine un sommet actif u, Δ est encore une distance estimée relative à $f.a$.*

Démonstration. On a toujours $\Delta(p) = 0$. Pour tous les arcs $x \to_{f.a} y$ de $G_{f.a}$ qui étaient déjà dans G_f, on a $\Delta(x) \leq \Delta(y) + 1$, car Δ est une distance estimée relative à f. On examine donc les arcs $x \to_{f.a} y$ de $G_{f.a}$ tels que $x \to_f y$ n'était pas un arc de G_f.

Si a est représentant conforme de $u \to v$, comme $f.a$ et f ne diffèrent qu'en l'arc $u \to v$, tous les arcs de G_f se retrouvent dans $G_{f.a}$, sauf éventuellement $u \to_f v$, qui disparaît si la poussée $(f, \Delta) \rightsquigarrow (f.a, \Delta)$ est saturante et $r_f(a) > 0$, par la remarque 9.65. Le seul arc supplémentaire dans $G_{f.a}$ est alors le conjoint $v \to_{f.a} u$. Mais a est admissible, donc $\Delta(u) = \Delta(v)+1$, et donc $\Delta(v) = \Delta(u)-1 \leq \Delta(v) + 1$. On raisonne de façon similaire si a est représentant non conforme (de $v \to u$). □

On a l'analogue suivant du lemme 9.27.

Lemme 9.67 *Soit f un préflot sur un réseau (G, s, p), et Δ une distance estimée relative à f. Pour tout arc admissible a du graphe d'écart G_f, d'origine un sommet actif u, l'ensemble des arcs du graphe d'admissibilité $G_{f.a,\Delta}$ est inclus dans l'ensemble des arcs du graphe d'admissibilité $G_{f,\Delta}$, et strictement inclus si la poussée $(f, \Delta) \rightsquigarrow (f.a, \Delta)$ est saturante.*

Démonstration. Par la remarque 9.65, $G_{f.a}$ est obtenu à partir de G_f en effaçant l'arc $a\colon u \to_f v$ si et seulement si la poussée est saturante, et en ajoutant possiblement son conjoint, mais aucun autre arc. Il suffit maintenant de vérifier que le conjoint de a n'est pas admissible : comme a est admissible, $\Delta(u) = \Delta(v) + 1$, donc $\Delta(v) \neq \Delta(u) + 1$. □

On a tout intérêt à effectuer des poussées saturantes, et le lemme 9.67 montre qu'on ne pourra pas en faire plus de $2m$ consécutives.

Définition 9.68 *Soit f un préflot sur un réseau (G, s, p), et Δ une distance estimée relative à f. Un sommet u est* bloqué *dans la configuration (f, Δ) si et seulement s'il est actif pour f et n'est l'origine d'aucun arc admissible dans $G_{f,\Delta}$.*

bloqué

Lorsqu'un sommet est bloqué, on peut le débloquer par une mise à jour de Δ, c'est-à-dire un *réétiquetage*.

Définition 9.69 (Réétiquetage) *Soit (G, s, p) un réseau. Une* transition de réétiquetage *est de la forme $(f, \Delta) \rightsquigarrow (f, u.\Delta)$, où f est un flot sur (G, s, p), Δ est une distance estimée relative à f, u est un sommet bloqué dans la configuration (f, Δ), et $u.\Delta$ est définie par :*

transition de réétiquetage

- $(u.\Delta)(u) \stackrel{def}{=} \min\{\Delta(v) + 1 \mid u \to_f v\}$ *si cet ensemble est non vide, n sinon ;*
- $(u.\Delta)(v) \stackrel{def}{=} \Delta(v)$ *pour tout sommet $v \neq u$.*

Le lemme suivant est l'analogue du lemme 9.30.

Lemme 9.70 *Dans les conditions de la définition 9.69, $u.\Delta$ est une distance estimée relative au préflot f. De plus, $(u.\Delta)(v) \geq \Delta(v)$ pour tout sommet v, l'inégalité étant stricte si $u = v$.*

Démonstration. Comme u est bloqué, $(u.\Delta)(u) > \Delta(u)$. En effet, sinon, on aurait $(u.\Delta)(u) \leq \Delta(u)$.

L'algorithme de Goldberg-Tarjan et Karzanov consiste à itérer les transitions de poussée et les transitions de réétiquetage, partant de (f_0, Δ_0), où f_0 est le préflot du lemme 9.60 et Δ_0 est la fonction calculée à la table 19, jusqu'à ce que l'on ne puisse plus opérer ni poussée ni réétiquetage.

Il est correct (s'il termine) pour une raison simple : une configuration (f, Δ) à laquelle on ne peut appliquer ni poussée ni réétiquetage est tel que f est un flot maximum. Ceci se démontre en deux lemmes.

Lemme 9.71 *Soit (G, s, p) un réseau à n sommets, et $(f_0, \Delta_0) \rightsquigarrow (f_1, \Delta_1) \rightsquigarrow \cdots \rightsquigarrow (f_k, \Delta_k)$ une suite de transitions de poussée ou de réétiquetage, où f_0 est le préflot du lemme 9.60 et Δ_0 est la fonction calculée par la procédure* `init-delta` *de la table 19, c'est-à-dire la fonction $\min(d(\_, p), n)$.*

Pour tout $i \in \{0, 1, \cdots, k\}$, on a $\Delta_i(s) = n$.

Démonstration. On démontre que $\Delta_i(s) = n$ par récurrence sur i. Lorsque i vaut 0, on a $f_0(s,u) = c_{\max}(s,u)$ pour tout arc $s \to u$ de G d'origine s, et il n'y a donc pas d'arc conforme $s \to_{f_0} u$ dans le graphe d'écart G_{f_0}. Il n'y a pas non plus d'arc non conforme $s \to_{f_0} u$, car sinon il y aurait un arc $u \to s$ (ce qui en soi n'est pas exclu) dans G avec $f_0(u,s) > 0$. Or, comme G n'a pas de boucle, $u \neq s$, et par définition on a en fait $f_0(u,s)$. Il n'y a donc aucun arc d'origine s dans le graphe d'écart. Dans ces conditions, on a $\Delta_0(s) = n$, car $\Delta_0(s) = \min(d(s,p), n) = \min(+\infty, n) = n$.

Supposons que $\Delta_i(s) = n$, avec $0 \leq i < k$. Si la transition $(f_{i-1}, \Delta_{i-1}) \rightsquigarrow (f_i, \Delta_i)$ est une transition de poussée, alors $\Delta_i = \Delta_{i-1}$, donc $\Delta_i(s) = n$ par hypothèse de récurrence. Si c'est une transition de réétiquetage, disons $\Delta_i = u.\Delta_{i-1}$, et alors u est un sommet bloqué, donc actif (définition 9.68), donc différent de s (définition 9.61). Mais Δ_{i-1} et $u.\Delta_{i-1}$ coïncident sur tous les sommets différents de u, donc $\Delta_i(s) = (u.\Delta_{i-1})(s) = \Delta_{i-1}(s) = n$. □

Remarque 9.72 *On peut en fait montrer que le puits p n'est accessible depuis s dans* aucun *des graphes d'écart G_{f_i} (ce qui implique $\Delta_i(s) = n$). La situation est donc duale de celle des algorithmes de Ford-Fulkerson et Dinic-Edwards-Karp, où le puits p était* toujours *accessible depuis s, sauf à la dernière étape.*

On remarque aussi que f_i est un flot à toute étape des algorithmes de Ford-Fulkerson et Dinic-Edwards-Karp, qui ne sera maximal qu'à la dernière étape, alors que dans l'algorithme de Goldberg-Tarjan et Karzanov, le préflot f_i n'est pas un flot, sauf à la dernière étape où il sera automatiquement maximum, comme nous l'observons ci-dessous.

On aboutit ainsi au théorème de correction partielle suivant.

Théorème 9.73 *Soit (G,s,p) un réseau à n sommets, et $(f_0, \Delta_0) \rightsquigarrow (f_1, \Delta_1) \rightsquigarrow \cdots \rightsquigarrow (f_k, \Delta_k)$ une suite de transitions de poussée ou de réétiquetage, où f_0 est le préflot du lemme 9.60 et Δ_0 est la fonction calculée par la procédure* `init-delta` *de la table 19, c'est-à-dire la fonction* $\min(d(\_, p), n)$.

Si (f_k, Δ_k) est une configuration stoppée*, c'est-à-dire si aucune transition ni de poussée ni de réétiquetage ne peut lui être appliquée, alors f_k est un flot de valeur maximum de (G,s,p).*

configuration stoppée

Démonstration. Comme aucune transition de poussée ne peut être appliquée à (f_k, Δ_k), tout sommet actif de G pour f_k est bloqué. Comme aucune transition de réétiquetage ne peut être appliquée à (f_k, Δ_k), il n'y a pas de sommet bloqué. Il n'y a donc aucun sommet actif pour f_k, autrement dit $\mathrm{e}_{f_k}(u) = 0$ pour tout sommet u de G. Il s'ensuit que f_k est un flot.

Par le lemme 9.71, $\Delta_k(s) = n$. Comme f_k est un flot, on peut appliquer le corollaire 9.22, et f_k est donc un flot de valeur maximum. □

La question de la terminaison, et de la complexité de l'algorithme, est un peu plus difficile. Nous allons montrer que les valeurs des distances estimées

sont majorées par $2n-1$ au cours de toute suite de transitions, ce qui nous permettra ensuite de majorer le nombre de poussées saturantes, et enfin le nombre de poussées non saturantes.

Le lemme suivant est implicite dans la démonstration du lemme 3.7 de la section 8.3.3 du BBC, page 267. Nous en donnons aussi une démonstration plus simple, adaptée du lemme 5 de la section 6.2.3 du livre de U. Brandes, S. Cornelsen, M. Badent et N. Indlekofer, *Design and Analysis of Algorithms*, Universität Konstanz, WS 2010/11.

Lemme 9.74 *Soit f un préflot sur un réseau (G, s, p). Pour tout sommet actif u pour f, il existe un chemin $\mu\colon s \to^* u$ dans G qui est* positif*, c'est-à-dire tel que pour tout arc e de μ, $f(e) > 0$; de façon équivalente, le chemin $\mu^{op}\colon u \to_f^* s$ obtenu en prenant les représentants non conformes de chaque arc de μ est un chemin du graphe d'écart G_f.* positif

Démonstration. Généralisons la notion d'excès, et pour tout sous-ensemble E de $S \setminus \{s\}$, où S est l'ensemble des sommets de G, posons $\mathrm{e}_f(E) \stackrel{\text{def}}{=} \mathrm{in}(f, E) - \mathrm{out}(f, E)$. En particulier, $\mathrm{e}_f(\{u\}) = \mathrm{e}_f(u)$ pour tout sommet $u \neq s$.

Nous avons d'abord le **Fait 1 :** pour tout $E \subseteq S \setminus \{s\}$, $\mathrm{e}_f(E) = \sum_{u \in E} \mathrm{e}_f(u)$. En effet,

$$\begin{aligned}
\sum_{u \in E} \mathrm{e}_f(u) &= \sum_{u \in E} \left(\sum_{v/v \to u} f(v \to u) - \sum_{w/u \to w} f(u \to w) \right) \\
&= \sum_{e\colon v \to u \in E} f(e) - \sum_{e\colon u \in E \to w} f(e) \\
&= \underbrace{\sum_{e\colon v \notin E \to u \in E} f(e) - \sum_{e\colon u \in E \to w \notin E} f(e)}_{=\mathrm{e}_f(E)} \\
&\quad + \underbrace{\sum_{e\colon v \in E \to u \in E} f(e) - \sum_{e\colon u \in E \to w \in E} f(e)}_{=0}.
\end{aligned}$$

Ensuite, nous démontrons le **Fait 2 :** pour tout $E \subseteq S \setminus \{s\}$, $\mathrm{e}_f(E) \geq 0$, et l'inégalité est stricte s'il existe un sommet u actif dans E. C'est une conséquence directe du Fait 1, puisque $\mathrm{e}_f(u) \geq 0$ (car f est un *préflot*) pour tout sommet u, l'inégalité étant stricte si u est actif.

Supposons maintenant que u soit un sommet actif pour f, et qu'il n'y ait aucun chemin de u à s par des arcs non conformes dans le graphe d'écart G_f. Soit E l'ensemble des sommets accessibles depuis u par des arcs non conformes dans G_f. Comme u est actif et dans E, par le Fait 2, $\mathrm{e}_f(E) > 0$.

On observe que pour tout arc $e\colon v \notin E \to w \in E$ dans G, $f(e) = 0$. En effet, si on avait $f(e) >$, e aurait un représentant non conforme $w \to_f v$ dans G_f, et comme $w \in E$, on aurait aussi $v \in E$, ce qui est impossible.

On en déduit que $\mathrm{in}(f,E) = \sum_{e:\, v\notin E \to w\in E} f(e) = 0$. Donc $\mathrm{e}_f(E) = -\mathrm{out}(f,E) \leq 0$, ce qui contredit l'inégalité $\mathrm{e}_f(E) > 0$ démontrée plus haut, au Fait 2. □

Corollaire 9.75 *Soit (G,s,p) un réseau à n sommets, et $(f_0,\Delta_0) \rightsquigarrow (f_1,\Delta_1) \rightsquigarrow \cdots \rightsquigarrow (f_k,\Delta_k)$ une suite de transitions de poussée ou de réétiquetage, où f_0 est le préflot du lemme 9.60 et Δ_0 est la fonction $\min(d(\_,p),n)$.*

Pour tout $i \in \{0,1,\cdots,k-1\}$, pour tout sommet actif u pour f_i, $\Delta_i(u) \leq 2n-1$. Par conséquent, le nombre de transitions $(f_{i-1},\Delta_{i-1}) \rightsquigarrow (f_i,\Delta_i)$ $(1 \leq i \leq k)$ qui sont des transitions de réétiquetage est d'au plus $(n-1)(2n-1)$.

Démonstration. Par le lemme 9.74, il existe un chemin $\mu\colon s = x_0 \to x_1 \to \cdots \to x_m = u$ positif dans G, et donc un chemin $\mu^{op}\colon u = x_m \to_{f_i} \cdots \to_{f_i} x_1 \to_{f_i} x_0 = s$ formé d'arcs non conformes dans G_{f_i}. On peut le choisir élémentaire, par le lemme 4.3 appliqué au sous-graphe couvrant de G formé des arcs e tels que $f(e) > 0$. Donc $m \leq n-1$. Par définition d'une distance estimée, appliquée aux arcs non conformes de μ^{op}, $\Delta_i(u) \leq \Delta_i(s) + m \leq \Delta_i(s) + n - 1$. Or par le lemme 9.71, $\Delta_i(s) = n$.

Pour ce qui est du nombre de transitions de réétiquetage, chaque transition $(f_{i-1},\Delta_{i-1}) \rightsquigarrow (f_i,\Delta_i)$ est telle que $\Delta_i \geq \Delta_{i-1}$. C'est évident s'il s'agit d'une transition de poussée, car dans ce cas $\Delta_i = \Delta_{i-1}$, et ceci provient du lemme 9.70 s'il s'agit d'une transition de réétiquetage. De plus, dans ce denier cas, on a $\Delta_i(u) > \Delta_{i-1}(u)$ pour un unique sommet u. De plus, ce sommet ne peut pas être égal à s, puisque $\Delta_i(s) = \Delta_{i-1}(s) = n$ par le lemme 9.71. Donc la somme $\sum_{u\in S\setminus\{s\}} \Delta_i(u)$ augmente d'au moins 1 à chaque transition de réétiquetage, et ne peut pas dépasser $\sum_{u\in S\setminus\{s\}}(2n-1) = (n-1)(2n-1)$. □

Proposition 9.76 *Sous les conditions du corollaire 9.75, au plus $2m(n-1)$ des transitions sont des transitions de poussée saturantes.*

Démonstration. Supposons que $(f_{i-1},\Delta_{i-1}) \rightsquigarrow (f_i,\Delta_i)$ soit une opération de poussée saturante, effectuant sur le sommet actif u et l'arc admissible $a\colon u \to_{f_{i-1}} v$. Donc $f_i = f_{i-1}.a$ et $\Delta_i = \Delta_{i-1}$, et par la remarque 9.65, l'arc a disparaît, c'est-à-dire qu'il n'y a pas d'arc $u \to_{f_i} v$ dans G_{f_i}. Son conjoint $v \to_{f_i} u$ est alors présent dans G_{f_i} (car $r_{f_{i-1}}(a) > 0$).

La seule façon qu'a l'arc a de réapparaître plus tard, disons lors d'une transition $(f_{j-1},\Delta_{j-1}) \rightsquigarrow (f_j,\Delta_j)$ avec $i < j < k$, est si cette transition est une poussée sur l'arc conjoint de a, disons à la transition m, avec $i < m < j$. Mais on ne peut opérer de poussée sur ce conjoint de a que s'il est admissible, autrement dit si $\Delta_{m-1}(v) = \Delta_{m-1}(u) + 1$. Or $\Delta_i(v) = \Delta_i(u) - 1$, puisque l'arc a était admissible lors de la transition i. De même, on ne peut effectuer de transition de poussée à la tradition i que si $\Delta_{j-1}(v) = \Delta_{j-1}(u) - 1$. Par lemme 9.70, et parce que les valeurs de distances estimées restent inchangées lors des transitions de poussée,

on a $\Delta_i \leq \Delta_{m-1} \leq \Delta_{j-1}$, donc :

$$\begin{aligned}\Delta_{j-1}(v) &= \Delta_{j-1}(u) - 1 \\ &\geq \Delta_{m-1}(u) - 1 \\ &= \Delta_{m-i}(v) - 2 \\ &\geq \Delta_{i-1}(v) - 2.\end{aligned}$$

Entre deux poussées saturantes sur un arc a de u à v, la distance estimée de v a donc augmenté d'au moins 2. Elle ne peut pas dépasser $2n-1$ par le corollaire 9.75, donc il y a au plus $n-1$ transitions de poussée saturante par arc a du graphe d'écart. Comme il y au plus $2m$ arc dans le graphe d'écart (pas m), ceci forme au plus $2m(n-1)$ transitions de poussée saturante. □

La proposition 9.78 ci-dessous sert à majorer le nombre de transitions de poussée non saturantes. C'est le lemme 3.9, section 8.3.3, page 268 du BBC, à quelques changements près sur les majorations. Elle repose sur une analyse de l'évolution des ensembles des sommets actifs lors d'une transition de poussée, saturante ou non, que nous explicitons maintenant.

Lemme 9.77 *Soit $(f, \Delta) \rightsquigarrow (f.a, \Delta)$ une transition de poussée, où $a\colon u \to_{r_f} v$. Alors :*

1. *u est actif pour f ;*
2. *v est actif pour $f.a$, sauf si $v = s$ ou si $v = p$;*
3. *pour tout sommet $w \neq u, v$, w est actif pour f si et seulement s'il l'est pour $f.a$;*
4. *si la poussée est non saturante, alors u n'est pas actif pour $f.a$.*

Démonstration. 1. Par définition d'une transition de poussée.

2. Il suffit de démontrer que $\mathrm{e}_{f.a}(v) > 0$. Si a est représentant conforme d'un arc $u \to v$ de G,

$$\begin{aligned}\mathrm{e}_{f.a}(v) &= \sum_{x/x\to v} (f.a)(x,v) - \sum_{y/v\to y} (f.a)(v,y) \\ &= \sum_{x\neq u/x\to v} f(x,v) + f(u,v) + \epsilon - \sum_{y/v\to y} f(v,y) \\ &= \mathrm{e}_f(v) + \epsilon,\end{aligned}$$

où $\epsilon \stackrel{\text{def}}{=} \min(\mathrm{e}_f(u), c_{\max}(u,v) - f(u,v))$, par la première partie de la définition 9.62. Or $\mathrm{e}_f(u) > 0$ car u est actif pour f, et $c_{\max}(u,v) - f(u,v) > 0$ puisque l'arc a est conforme, donc l'arc $u \to v$ est libre par rapport à f. Donc $\epsilon > 0$, d'où l'on déduit $\mathrm{e}_{f.a}(v) > 0$.

Si a est représentant non conforme d'un arc $v \to u$ de G,

$$\begin{aligned}
\mathrm{e}_{f.a}(v) &= \sum_{x/x\to v} (f.a)(x,v) - \sum_{y/v\to y} (f.a)(v,y) \\
&= \sum_{x/x\to v} f(x,v) - \sum_{y\neq u/v\to y} f(v,y) - f(v,u) + \epsilon \\
&= \mathrm{e}_f(v) + \epsilon,
\end{aligned}$$

où $\epsilon \stackrel{\text{def}}{=} \min(\mathrm{e}_f(u), f(v,u))$ par la deuxième partie de la définition 9.62. Or, comme a est représentant non conforme de $v \to u$, l'arc $v \to u$ est passant par rapport à f, autrement dit $f(v,u) > 0$, et d'autre part $\mathrm{e}_f(u) > 0$ puisque u est actif pour f. Donc $\epsilon > 0$, ce qui implique $\mathrm{e}_{f.a}(v) > 0$ de nouveau.

3. Soit w un sommet différent de u et de v. Les excès de $f_i = f_{i-1}.a$ et de f_{i-1} en w sont identiques, car ils sont définis comme des différences de sommes portant sur des valeurs de flots en des arcs d'origine ou d'extrémité w, donc différents de l'unique arc ($u \to v$ ou $v \to u$) dont la valeur de flot change entre f_{i-1} et f_i. Donc, w est actif pour f_i si et seulement s'il est actif pour f_{i-1}.

4. On suppose maintenant la poussée non saturante, autrement dit $\mathrm{e}_f(u) < r_f(a)$. Si a est représentant conforme de $u \to v$, alors :

$$\begin{aligned}
\mathrm{e}_{f.a}(u) &= \sum_{x/x\to u} (f.a)(x,u) - \sum_{y/u\to y} (f.a)(u,y) \\
&= \sum_{x/x\to u} f(x,u) - \sum_{y\neq v/u\to y} f(u,y) - f(u,v) - \epsilon \\
&= \mathrm{e}_f(u) - \epsilon,
\end{aligned}$$

où $\epsilon \stackrel{\text{def}}{=} \min(\mathrm{e}_f(u), c_{\max}(u,v) - f(u,v)) = \min(\mathrm{e}_f(u), r_f(a))$, par la première partie de la définition 9.62 pour $f.a$, et donc $\epsilon = \mathrm{e}_f(u)$, puisque $\mathrm{e}_f(u) < r_f(a)$; et ceci implique $\mathrm{e}_{f.a}(u) = 0$. De façon similaire, si a est représentant non conforme de $v \to u$, alors :

$$\begin{aligned}
\mathrm{e}_{f.a}(u) &= \sum_{x/x\to u} f.a(x,u) - \sum_{y/u\to y} f.a(u,y) \\
&= \sum_{x\neq v/x\to u} f(x,u) + f(v,u) - \epsilon - \sum_{y\neq u/u\to y} f(u,y) - f(v,u) + \epsilon \\
&= \mathrm{e}_f(u) - \epsilon,
\end{aligned}$$

où $\epsilon \stackrel{\text{def}}{=} \min(\mathrm{e}_f(u), f(v,u)) = \min(\mathrm{e}_f(u), r_f(a))$ par la deuxième partie de la définition 9.62. Comme $\mathrm{e}_f(u) < r_f(a)$, $\epsilon = \mathrm{e}_f(u)$, donc $\mathrm{e}_{f.a}(u) = 0$. □

Proposition 9.78 *Sous les conditions du corollaire 9.75, au plus $8n^2m$ des transitions sont des transitions de poussée non saturantes.*

Démonstration. On définit une fonction *potentiel* ϕ en posant $\phi(f, \Delta)$ égal à la somme des distances estimées $\Delta(u)$ sur tous les sommets u actifs pour f. On définit ensuite la *variation de potentiel* $\delta\phi_i$ à la transition i par :

$$\delta\phi_i \stackrel{\text{def}}{=} \phi(f_i, \Delta_i) - \phi(f_{i-1}, \Delta_{i-1}).$$

Pour fixer les idées, on note que $\phi(f_0, \Delta_0)$ est une quantité positive, et que si (f_k, Δ_k) était une transition stoppée (ce que nous ne supposons pas, au passage), on aurait $\phi(f_k, \Delta_k) = 0$; on peut donc s'attendre à ce que $\delta\phi_i$ soit usuellement négatif.

On a $\sum_{i=1}^{k} \delta\phi_i = \phi(f_k, \Delta_k) - \phi(f_0, \Delta_0)$. Il y au plus n sommets, donc au plus n sommets actifs pour f_0, et s et p ne sont pas actifs par définition, donc il y a au plus $n-2$ sommets actifs u pour f_0. Pour chacun, $\Delta_0(u) = \min(d(u,p), n) \leq n$, donc $\phi(f_0, \Delta_0) \leq n(n-2)$. Comme $\phi(f_k, \Delta_k) \geq 0$, on en déduit que :

$$\sum_{i=1}^{k} \delta\phi_i \geq -n(n-2). \tag{3}$$

Considérons maintenant la transition i, pour un i quelconque entre 1 et k, et examinons le type de cette transition.

Réétiquetage. Si la transition i est une transition de réétiquetage, disons sur u (autrement dit, $\Delta_i = u.\Delta_{i-1}$), alors $\Delta_i(v) = \Delta_{i-1}(v)$ pour tout sommet $v \neq u$, et les sommets actifs pour f_i sont les mêmes que pour f_{i-1}, puisque $f_i = f_{i-1}$. Donc $\delta\phi_i = \Delta_i(u) - \Delta_{i-1}(u)$. Ceci est supérieur ou égal à 1, mais ce qui nous intéresse davantage est que $\delta\phi_i \leq \Delta_i(u) \leq 2n-1$, par le corollaire 9.75.

Poussée. Par le lemme 9.77, points 3, et comme $\Delta_i = \Delta_{i-1}$, les termes $\Delta_i(w)$ et $\Delta_{i-1}(w)$ s'annulent pour tout sommet $w \neq u, v$ dans la somme définissant $\delta\phi_i$. Donc $\delta\phi_i = [\Delta_i(u)+]\Delta_i(v) - \Delta_{i-1}(u)[-\Delta_{i-1}(v)]$ si $v \neq s, p$ et $\delta\phi_i = [\Delta_i(u)] - \Delta_{i-1}(u)[-\Delta_{i-1}(v)]$ sinon, où un terme entre crochets $[a]$ signifie a ou une absence de terme ; l'un des deux termes entre crochets peut être présent ou absent sans que l'autre le soit. Le terme $\Delta_i(v)$ est présent si $v \neq s, p$ par le point 2 du lemme 9.77, et le terme $-\Delta_{i-1}(u)$ $(= \Delta_i(u))$ est présent par le point 1 de ce même lemme.

Poussée saturante. On majore $\delta\phi_i$ par $\Delta_i(u) + \Delta_i(v) - \Delta_{i-1}(u) = \Delta_i(v) \leq 2n-1$, grâce au corollaire 9.75.

Poussée non saturante. Alors u n'est plus actif pour f_i, par le point 4 du lemme 9.77. Donc $\delta\phi_i = \Delta_i(v) - \Delta_{i-1}(u)[-\Delta_{i-1}(v)]$ si $v \neq s, p$ et $\delta\phi_i = -\Delta_{i-1}(u)$ $[-\Delta_{i-1}(v)]$ sinon. Comme a est un arc admissible par rapport à la distance estimée Δ_{i-1} $(= \Delta_i)$, on a $\Delta_i(u) = \Delta_i(v) + 1$, donc $\delta\phi_i \leq -1$ si $v \neq s, p$. Si $v = s$ ou $v = p$, on a $\delta\phi_i \leq -\Delta_{i-1}(u)$, qui est aussi inférieur ou égal à -1 : comme u est actif, par définition $u \neq p$, donc la distance de u à p est supérieure ou égale à 1 ; or cette distance de u à p vaut $\Delta_0(u)$, et comme les distances estimées ne font qu'augmenter (lemme 9.70 pour les transitions de réétiquetage, pas de variation pour les transitions de poussée), $\Delta_{i-1}(u) \leq 1$.

Soit N le nombre de transitions de poussée non saturantes dans la suite de transitions données. On a :

— au plus $(n-1)(2n-1)$ réétiquetages (corollaire 9.75), chacune de $\delta\phi_i \leq 2n-1$;

— au plus $2m(n-1)$ poussées saturantes (proposition 9.78), chacune de $\delta\phi_i \leq 2n-1$;

— N poussées non saturantes, chacune de $\delta\phi_i \leq -1$.

Donc $\sum_{i=1}^{k} \delta\phi_i \leq (n-1)(2n-1)(2n-1) + 2m(n-1)(2n-1) + N \times (-1)$. Or par (3), $\sum_{i=1}^{k} \delta\phi_i \geq -n(n-2)$, donc :

$$\begin{aligned}
N &\leq (n-1)(2n-1)(2n-1) + 2m(n-1)(2n-1) + n(n-2) \\
&= (4n^3 - 8n^2 + 5n - 1) + (4n^2m - 6nm + 2m + (n^2 - 2n) \\
&= 4n^3 - 7n^2 + 3n - 1 + 4n^2m - 6nm + 2m \\
&\leq 8n^2m - 7n^2 + 5m - 1 - 6nm \\
&\qquad \text{puisque } n \leq m \text{ (remarque 9.38), donc} 4n^3 \leq 4n^2m \\
&\leq 8n^2m - 7n^2 - m - 1 \\
&\qquad \text{puisque } n \geq 1 \text{ (et même } n \geq 2, \text{ car } s \neq p), \text{ donc } 6nm \geq 6m \\
&\leq 8n^2m.
\end{aligned}$$

□

Récapitulons. L'algorithme, pour l'instant abstrait, de Goldberg-Tarjan et Karzanov commence par une phase d'initialisation, en temps $O(n+m) = O(m)$. Il effectue ensuite :

— au plus $2n-1$ transitions de réétiquetage par sommet u (corollaire 9.75), qui prennent un temps majoré par quelque chose de proportionnel au nombre de successeurs de u dans le graphe d'écart courant plus une constante, c'est-à-dire à $\#\texttt{succ}(u) + \#\texttt{pred}(u) + 1$; le coût total de ces transitions est donc majoré par quelque chose de proportionnel à $(2n-1)\sum_{u \in S}(\#\texttt{succ}(u) + \#\texttt{pred}(u) + 1) = (2n-1)(2m+n) = O(nm)$, puisque $n \leq m$ par la remarque 9.38 ;

— au plus $2m(n-1)$ poussées saturantes (proposition 9.78) et au plus $8n^2m$ poussées saturantes (proposition 9.78), chacune en temps constant, pour un coût total en $O(n^2m)$ (oui, ce sont les poussées non saturantes qui coûtent le plus dans cet algorithme).

Il ne reste qu'à compter le temps pour détecter les sommets actifs et les arcs admissibles, et pour cela nous devons dire comment ils sont découverts.

Une solution relativement simple consiste à maintenir au cours du calcul un tableau `exces` qui à chaque sommet $u \neq s, p$ associe l'excès $\mathrm{e}_f(u)$ du flot courant f en u (oui, c'est un invariant), et un ensemble des sommets actifs. L'initialisation de ce tableau, ainsi que le calcul du préflot initial f_0, est montrée à la table 34. On peut représenter l'ensemble des sommets actifs par un arbre équilibré, de sorte

```
   fun init-preflot (G as (S, succ, pred), s, cmax, n) =
1      let exces = new(n), f = new-flow(n) in
2      for each u ∈ S do
3               (exces[u] := 0;
4                for each v ∈ succ(u) do
5                        f(u,v) := 0);
6      for each v ∈ succ(s) do
7               (f(s,v) := cmax(s,v);
8                exces[v] := cmax(s,v));
9      return f, exces;
```

TABLE 34 – Le calcul du préflot et des excès initiaux

que les opérations d'ajout, de retrait et de sélection d'un sommet actif se fassent en temps $O(\log n)$, mais ceci mènerait à une complexité globale de l'algorithme en $O(n^2 m \log n)$.

Karzanov a eu l'idée que l'on pouvait toujours choisir un sommet de distance estimée maximale, à condition d'utiliser la structure de donnée adéquate, en temps constant. Cette structure de données adéquates est une partition de l'ensemble des sommets par distances estimées, et remonte à Dinic (1970). De plus, ce choix va aussi diminuer le nombre de poussées non saturantes, qui forment le goulot d'étranglement de l'algorithme. Montrons-le tout de suite, avant de passer à l'implémentation.

Proposition 9.79 *Soit (G, s, p) un réseau à n sommets, et $(f_0, \Delta_0) \rightsquigarrow (f_1, \Delta_1) \rightsquigarrow \cdots \rightsquigarrow (f_k, \Delta_k)$ une suite de transitions de poussée ou de réétiquetage, où f_0 est le préflot du lemme 9.60 et Δ_0 est la fonction $\min(d(\_, p), n)$. Supposons de plus que la stratégie de sélection de la transition à effectuer à chaque étape soit de sélectionner un sommet actif u de distance estimée maximale, puis d'effectuer toutes les poussées sur les arcs admissibles d'origine u dans le graphe d'écart courant.*

Alors le nombre de transitions de poussée non saturantes est majoré par $2n(n^2 + 1)$.

Ceci est en $O(n^3)$, et améliore donc la majoration par $8n^2m$ obtenue à la proposition 9.78, puisque $n \leq m$ par la remarque 9.38.

Démonstration. [Esquisse] Fixons un sommet u. Sur un segment $[i, j]$ de transitions entre deux réétiquetages sur u (ou avant le premier, ou après le dernier), on ne peut faire qu'au plus une poussée non saturante sur un arc admissible d'origine u ; autrement dit, si on effectue une poussée non saturante sur un arc admissible d'origine u à une transition m avec $i < m < j$, alors les autres poussées sur un arc d'origine u (pas nécessairement le même) à des transitions entre $m + 1$ et $j - 1$ seront saturantes. En effet, une poussée non saturante ramène l'excès du flot courant en u à 0. Le sommet u perd alors son statut de sommet actif. Soit

```
   fun init-partition (S, exces, Delta, s, p, n) =
1      dmax := -1;
2      let partition = new(2n) in
3          for i=0..2n-1 do
4              partition[i] := [];
5          for each u ∈ S do
6                    if u≠s and u≠p then
7                       let i=d[u] in
8                           (partition[i] := u::partition[i];
9                            if exces(u)>0 and i>dmax
10                              then dmax := i);
11     return partition, dmax;
```

TABLE 35 – Le calcul des partitions de Dinic-Karzanov à partir d'une distance estimée

d la distance estimée de u à la transition m — ou à n'importe quelle transition entre $i+1$ et $j-1$, puisqu'aucun réétiquetage n'y est effectué. Comme toutes les poussées aux transitions $m+1, \ldots, j-1$ sont effectuées sur des sommets actifs v de distances estimées maximales, la première est effectuée sur un sommet actif v_1 de distance estimée strictement inférieure à d, donc $v_1 \neq u$, et donc l'excès du flot courant en u reste nul, puisqu'il est toujours propagé en direction de sommets distances estimées strictement inférieures (voir la définition d'un arc admissible), et u reste inactif. La deuxième poussée après celle sur v_1, si elle existe, est effectuée sur un sommet actif v_2 de distance estimée strictement inférieure à d elle aussi, donc $v_2 \neq u$, donc u reste inactif ; et ainsi de suite. Ceci montre donc qu'après une poussée non saturante sur (un arc admissible d'origine) u sur le segment $[i, j]$, plus aucune poussée ne sera effectuée sur u.

On ne fera donc qu'au plus $n-2$ poussées non saturantes entre deux réétiquetages (c'est le nombre de sommets différents de s et de p), ou avant le premier, ou après le dernier, donc au plus. Il y a au plus $(n-1)(2n-1)$ transitions de réétiquetage par le corollaire 9.75, donc au plus

$$\begin{aligned}(n-2)((n-1)(2n-1)+2) &= (n-2)(2n^2-3n+3)\\ &= 2n^3-7n^2+9n-6\\ &\leq 2n^3+2n\\ &\qquad \text{puisque } n \geq 1, \text{ donc } 9n-6 \leq 9n^2-6 \leq 9n^2\\ &= 2n(n^2+1)\end{aligned}$$

poussées non saturantes. □

La réalisation de cette idée consiste à maintenir un tableau `partition`, tel que pour tout $i \in \{0, 1, \cdots, 2n-1\}$, `partition`$[i]$ contienne une liste de tous les

```
   fun push-flow ((u, v), Delta, f, exces, epsilon)) =
1      if Delta[u]>Delta[v] (* cas conforme (pour un arc admissible) *)
2         then (f(u,v) := f(u,v)+epsilon;
3               exces[v] := exces[v]+epsilon);
4      else  (* cas non conforme (pour un arc admissible) *)
5              (f(u,v) := f(u,v)-epsilon;
6               exces[v] := exces[v]-epsilon);
```

TABLE 36 – Une poussée de flot, avec maintien du tableau `exces`

```
   fun select (partition, dmax) =
1      d := dmax;
2      while d≥0 do
3            if partition[d]=[]
4               then d := d-1
5            else let u::rest = partition[d] in
6                     (partition[d] := rest;
7                      return u, d);
8      raise Fini;
```

TABLE 37 – La sélection de sommet par la stratégie de Karzanov

sommets de distance estimée égale à i exactement (oui, c'est un autre invariant). Le calcul d'un tel tableau est montré à la table 35, où l'on calcule aussi la distance estimée maximale sur tous les sommets autres que la source `s` et le puits `p`, ou -1 s'il n'y en a pas. Un autre invariant de l'algorithme sera que `dmax` sera toujours supérieur ou égal à la distances estimée maximale d'un sommet actif.

On réalise la poussée d'une quantité `epsilon` le long d'un arc `u` $\rightarrow_f$ `v` du graphe d'écart par la procédure de la table 36, qui maintient l'invariant selon lequel `exces`$[x]$ est égale à l'excès du préflot `f` en tout sommet x. On notera qu'avec la stratégie de Karzanov, et comme on l'a noté au cours de la démonstration de la proposition 9.79, une poussée saturante sur (un chemin admissible d'origine) un sommet actif u, nécessairement de distance estimée maximale laissera actif, et donc la distance estimée maximale d'un sommet actif sera inchangée. Dans le cas d'une poussée non saturante, cette distance estimée maximale décroîtra. Cependant, nous ne mettrons à jour la variable `dmax` que de façon paresseuse, lors de la sélection du prochain sommet, voir la table 37.

La fonction `select` qui y est définie retourne non seulement un sommet `u` de plus grande distance estimée inférieure ou égal à `dmax`, mais aussi sa distance estimée `d`, tout en l'enlevant de la partition. Entre deux réétiquetages, le temps total des sélections effectuées par la fonction `select` est majoré par quelque

```
   fun relabel (G as (S, succ, pred), u, f, cmax, Delta, partition, n) =
1      new_d := n; (* sera la nouvelle distance estimée de u *)
2      for each v ∈ succ(u) do
3              if Delta[u]>Delta[v] and f(u,v)<cmax(u,v)
                                   and Delta[v]+1<new_d
4                 then new_d := Delta[v]+1
5      for each v ∈ pred(u) do
6              if Delta[u]>Delta[v] and f(v,u)>0
                                   and Delta[v]+1<new_d
7                 then new_d := Delta[v]+1
8      Delta[u] := new_d; (* mise à jour de la distance estimée de u *)
9      partition[new_d] := u::partition[new_d] ;
           (* on rajoute u à la bonne entrée de partition *)
10     return max(dmax, new_d) ;
```

TABLE 38 – L'opération de réétiquetage, avec calcul d'une nouvelle valeur de `dmax` et maintien de la table `partition`

chose de proportionnel à la somme sur tout d allant de $2n-1$ à 0 du cardinal de `partition`$[d]$ plus 1 (pour le temps pris pour faire décroître d), c'est-à-dire à $n+2n = 3n$. Le temps cumulé des sélections sur toute l'exécution de l'algorithme est donc majoré par $3n((n-1)(2n-1)+2)$, en utilisant le corollaire 9.75, qui est en $O(n^3)$.

La fonction de réétiquetage est montrée à la table 38. Sa pré-condition (informelle) est que `u` est un sommet actif, et que `partition` est un tableau de tous les sommets sauf la source `s`, le puits `p`, et `u` lui-même (qui aura été enlevé par le dernier appel à `select`), rangé par distances estimées. L'invariant selon lequel `partition` est un tableau de tous les sommets sauf la source `s` et le puits `p` est rétabli à la ligne 9, et une nouvelle valeur de `dmax`, garantie supérieure ou égale à toutes les distances estimées de sommets actifs, est retournée à la ligne 10. La complexité de `relabel` est majorée par quelque chose de proportionnel à #`succ(u)` + #`pred(u)` + 1, comme auparavant. Les lignes supplémentaires 9 et 10 ne coûtent qu'un temps supplémentaire constant.

L'algorithme final de Goldberg-Tarjan et Karzanov est montré à la table 39. Nous énonçons finalement la complexité de cet algorithme qui se déduit de nos réflexions faites plus haut, notamment la proposition 9.79 et le fait que le coût total des appels à `select` est en $O(n^3)$.

Théorème 9.80 *La complexité de l'algorithme de Goldberg-Tarjan et Karzanov est en $O(n^3)$.*

La correction du code de la table 39, par opposition à la correction de l'algorithme abstrait, est laissée en exercice, pour reprendre un poncif des textes de

```
  fun goldberg-tarjan-karzanov (G as (S, succ, pred), s, p, cmax, n) =
1     let f, exces = init-preflot (G, s, cmax, n) in
2     let Delta = init-delta (G, p, f, cmax, n) in
3     let partition = init-partition (S, exces, Delta, s, p, n) in
4     try while true do
5         let u, d = select (partition, dmax) in
6             (bloque := true;
7              for each v ∈ succ(u) do (* poussées sur arcs admissibles conformes *)
8                       if exces[u]>0 and Delta[u]=Delta[v]+1
                                      and f(u,v)<cmax(u,v)
9                          then (bloque := false;
10                              push-flow ((u, v), Delta, f, exces,
                                           min (exces[u], cmax(u,v)-f(u,v))));
11             for each v ∈ pred(u) do (* poussées sur arcs admissibles non conformes *)
12                      if exces[u]>0 and Delta[u]=Delta[v]+1
                                      and f(v,u)>0
13                         then (bloque := false;
14                              push-flow ((u, v), Delta, f, exces,
                                           min (exces[u], f(v,u));
15             if bloque (* sommet bloqué : on réétiquette *)
16                then dmax := relabel (G, u, f, cmax, Delta, partition, n));
17    match Fini => return f;
```

TABLE 39 – L'algorithme de Goldberg-Tarjan et Karzanov

mathématiques. Certains des invariants nécessaires ont déjà été énoncés au cours de la description de l'algorithme.

Nous terminons ici ces notes. L'algorithme de Goldberg-Tarjan et Karzanov fonctionne en réalité avec une complexité légèrement meilleure de $O(n^2\sqrt{m})$ (J. Cheriyan et S. N. Maheshwari, *Analysis of Preflow Push Algorithms for Maximum Network Flow*, SIAM J. Computing 18 :1057–1086, 1989). Goldberg et Tarjan ont ensuite observé que l'on pouvait implémenter la stratégie de Karzanov en effectuant des poussées simultanément sur plusieurs arcs, grâce à l'utilisation d'arbres dynamiques, obtenant ainsi une complexité en $O(nm \log(n^2/m))$. Il existe aussi une version à échelonnement de l'algorithme de Goldberg-Tarjan et Karzanov, de complexité $O(nm + n^2 \log C)$, dans laquelle ont sélectionne toujours un sommet dont l'excès est supérieur ou égal à 2^k, pour un entier k qui décroît au fur et à mesure, et parmi ces sommets, on en choisit un de distance estimée maximale comme plus haut : voir le BBC, section 8.3.5, page 270.