UFR d'Informatique
Paris 7 – Paris Diderot
Année 2010–2011

Notes de cours d'algorithmique – L3

François Laroussinie

Notes de cours d'algorithmique – L3

François Laroussinie
francois.laroussinie@liafa.jussieu.fr
Page web du cours : http://www.liafa.jussieu.fr/~francoisl/l3algo.html

2 novembre 2010

Mode d'emploi :
Ces notes de cours sont un complément aux cours et aux TD.

Extrait du Petit Robert : "**Complément :** Ce qui s'ajoute ou doit s'ajouter à une chose pour qu'elle soit complète. (...)"

Table des matières

A. Introduction

1 Rappel sur la complexité des algorithmes

Obj : évaluer l'efficacité d'un algorithme.

Dans la suite, on considère un algorithme A et on note $\mathcal{D}$ l'ensemble des données possibles : une exécution de A se fait sur une donnée $x \in \mathcal{D}$.

Définition 1 *Sur une donnée $x \in \mathcal{D}$ de taille n, l'algorithme A requiert un certain temps mesuré en nombre d'opérations élémentaires : $C_A(x)$.*

NB : on ne veut pas évaluer ce temps en minutes, secondes,... Il faut une notion **robuste** qui soit valide pour n'importe quel ordinateur, sans hypothèse sur le matériel, le compilateur, le langage de programmation, *etc.*

Qu'est-ce qu'une opération élémentaire ? C'est une opération qui prend un temps constant (ou plus exactement qui peut être considérée comme prenant un temps constant). On prendra par exemple : la multiplication scalaire pour un algorithme de produit de matrices, la comparaison de deux éléments pour un algorithme de tri, *etc.* Ce choix dépend de l'application, il est discutable et doit être motivé.

Définition 2 (Complexité dans le pire cas)

$$\mathcal{C}_A(n) \overset{def}{=} \max_{x.|x|=n} C_A(x)$$

: max des coûts de l'algorithme A sur toutes les données de taille n

Définition 3 (Complexité en moyenne) *Soit p une distribution de probabilités sur les données de taille n. On définit la complexité en moyenne de l'algorithme A sur les données de taille n selon p par :*

$$\mathcal{C}_A^{moy}(n) \overset{def}{=} \sum_{x.|x|=n} p(x) \cdot C_A(x)$$

Souvent on considère une distribution uniforme : $p(x) = \frac{1}{T(n)}$ où $T(n)$ est le nombre de données de taille n, c'est-à-dire $T(n) \overset{\text{def}}{=} |\{x \in \mathcal{D} \text{ tel que } |x| = n\}|$. Dans ce cas, on a : $\mathcal{C}_A^{moy}(n) \overset{\text{def}}{=} \frac{1}{T(n)} \cdot \sum_{x.|x|=n} C_A(x)$.

D'autres notions existent : complexité dans le meilleur cas, complexité en **espace**, complexité amortie, ...

1.1 Evaluer la complexité d'un algorithme

Obj : avoir un ordre de grandeur du nombre d'opérations que l'algorithme doit effectuer lorsque la taille du problème augmente.

On utilise les notations $O(\ )$, $\Omega(\ )$ et $\theta(\ )$ pour exprimer une majoration, une minoration ou un encadrement de l'ordre de grandeur :

- $O(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \exists c > 0, n_0 \geq 0 \text{ tq } 0 \leq f(n) \leq c \cdot g(n)\ \forall n \geq n_0\}$
  C'est donc l'ensemble des fonctions **majorées** par $c \cdot g(n)$.
- $\Omega(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \exists c > 0, n_0 \geq 0 \text{ tq } 0 \leq c \cdot g(n) \leq f(n)\ \forall n \geq n_0\}$
  C'est donc l'ensemble des fonctions **minorées** par $c \cdot g(n)$.
  $(f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(f(n)))$
- $\theta(g(n)) \stackrel{\text{def}}{=} \{f(n) \mid \exists c_1, c_2 > 0, n_0 \geq 0 \text{ tq } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\ \forall n \geq n_0\}$

On s'intéresse à quelques grandes "familles" de fonctions pour distinguer la complexité des algorithmes :

- les algorithmes sous-linéaires. Par exemple en $O(\log(n))$
  (exemple : la recherche dichotomique)
- les algorithmes linéaires : $O(n)$ (exemple : recherche du min/max dans une liste). Ou quasi-linéaires comme $O(n \cdot \log(n))$ (exemple : le tri d'un tableau)
- les algorithmes polynomiaux : $O(n^k)$

- les algorithmes exponentiels : $O(2^n)$ ou plus généralement en $O(2^{P(n)})$ où $P(n)$ est un polynôme.
- les algorithmes doublement-exponentiels $O(2^{2^n})$,...
- ...

Cette classification renvoie à de vraies différences :

- recherche dichotomique : la recherche d'un nom dans un annuaire comportant 1.000.000 de noms nécessite au plus 20 comparaisons si on applique l'algorithme dichotomique. (après une comparaison, l'espace de recherche n'est plus que de 500.000, après 2 il est de 250000, ...)
- tri d'un tableau : le tri d'un tableau de 100.000 nombres demande de l'ordre de 2 millions de comparaisons si on applique un algorithme efficace en $O(n \cdot \log(n))$... Mais il en nécessite de l'ordre de 10 milliards si on utilise un algorithme en $O(n^2)$.

N'oublions pas les ordres de grandeurs de toutes ces fonctions... [voir "Algorithmics, the spirit of computing", David Harel, Addison Wesley]

fct $\backslash n$	10	50	100	300	1000
$5n$	50	250	500	1500	5000
$n \cdot \log_2(n)$	33	282	665	2469	9966
n^2	100	2500	10000	90000	$10^6(7c)$
n^3	1000	125000	$10^6(7c)$	$27 \cdot 10^6(8c)$	$10^9(10c)$
2^n	1024	...$(16c)$	...$(31c)$	...$(91c)$	...$(302c)$
$n!$	$3.6 \cdot 10^6(7c)$	...$(65c)$	...$(161c)$	...$(623c)$	...!!!
n^n	$10 \cdot 10^9(11c)$	...$(85c)$	...$(201c)$	...$(744c)$	...!!!!

Notation : "(Xc)" signifie "s'écrit avec X chiffres en base 10".

A titre de comparaison : le nombre de protons dans l'univers comporte 79 chiffres... et le nombre de microsecondes depuis le big-bang a 24 chiffres !!

Un autre point de vue [voir "Algorithmics, the spirit of computing"] : supposons que l'on dispose d'un algorithme de complexité $f(n)$ qui permet de résoudre en une heure les instances de taille X d'un problème sur un ordinateur aujourd'hui. Alors :

- si $f(n) = n$, un ordinateur 100 fois plus rapide permettrait en une heure de résoudre les instances de taille $100X$; Et un ordinateur 1000 fois plus rapide permettrait de résoudre les instances de taille $1000 \cdot X$.
- si $f(n) = n^2$, un ordinateur 100 fois plus rapide permettrait de résoudre en une heure les instances de taille $10 \cdot X$; Et un ordinateur 1000 fois plus rapide permettrait de résoudre les instances de taille $31,6 \cdot X$ ($\sqrt{1000} \approx 31.6$).
- si $f(n) = 2^n$, un ordinateur 100 fois plus rapide permettrait de résoudre en une heure les instances de taille $X + 6,64$; Et un ordinateur 1000 fois plus rapide permettrait de résoudre les instances de taille $X + 9,97$.

Exercice : Pourquoi ?

En pratique, seule une petite portion des algorithmes sont utilisables... ceux de complexité polynomiale (avec des coefficients raisonnables).

Beaucoup des problèmes importants ne disposent que d'algorithmes dont la complexité est beaucoup trop élevée pour être utilisés en vrai !

On verra aussi que l'étude de la complexité permet de s'assurer qu'un algorithme est optimal (du point de vue de la complexité). Exemple (voir le cours de L2) : le tri.

B. Arbres binaires de recherche

Les ABR sont un type de données pour représenter un *dictionnaire* c'est à dire un ensemble de clés (ou un ensemble d'éléments accessibles via des clés) totalement ordonnées, muni des opérations "ajouter" (une clé), "rechercher", "supprimer".

Dans la suite on supposera que les clés sont des valeurs dans $\mathbb{N}$ mais ces clés peuvent être prises dans n'importe quel ensemble totalement ordonné.

1 Définitions

Définition 4 *Un* arbre binaire *(AB) est soit l'arbre vide, soit constitué d'une racine avec un couple (*i.e. *une paire ordonnée) d'arbres binaires appelés sous-arbre gauche et sous-arbre droit.*

Afin de décrire les algorithmes et d'évaluer leur complexité, on distingue les *noeuds externes* correspondant aux arbres vides, et les *noeuds internes* correspondant aux noeuds ayant des sous-arbres.

Exemple 1 *La figure 1 contient un exemple d'arbre binaire : les ◯ correspondent aux noeuds internes et les ■ aux noeuds externes.*

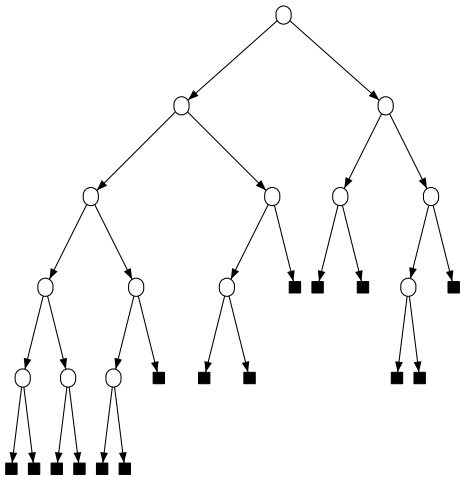

FIGURE 1 – Exemple d'arbre binaire.

Un arbre binaire *étiqueté* est un AB où l'on associe à chaque **noeud interne** un élément (appelé une **clé**).

Les opérations de base sur les AB sont :

- `est_vide`(`arbre` a) : `booléen` : teste si a est vide ou non ;
- `G`(`arbre` a) : `arbre` : renvoie le sous-arbre gauche de a (si a est non vide) ;
- `D`(`arbre` a) : `arbre` : renvoie le sous-arbre droit de a (si a est non vide) ;
- `val`(`arbre` a) : `clé` : donne la clé associée à la racine de a.

Les constructeurs sont :

- `ArbreVide`() : `arbre` – crée un arbre vide

– `Arbre`(`clé` x; `arbre` a_1, a_2) : `arbre` – crée une racine contenant x et avec a_1 et a_2 comme sous-arbre (g et d).

On peut aussi utiliser les fonctions suivantes pour modifier les champs d'un noeud interne :

– `FixerVal`(`clé` x; `arbre` a) : `arbre` – place x à la racine de a (ok si a est l'arbre vide).
– `FixerAG`(`arbre` a, g) : `arbre` – remplace le sous-arbre gauche de a par g.
– `FixerAD`(`arbre` a, d) : `arbre` – remplace le sous-arbre droit de a par d.

... mais on préférera écrire directement : `val`(a) := x, `G`(a) := g et `D`(a) := d.

Définition 5 *Un* Arbre Binaire de Recherche *A est un AB étiqueté tel que tout noeud interne s de A contient une clé x :*

– *supérieure (NB : ou égale si les clés ne sont pas toutes distinctes) à toutes les clés contenues dans le sous-arbre gauche de s ;*
– *inférieure strictement à toutes les clés contenues dans le sous-arbre droit de s.*

La figure 2 donne un exemple d'arbre binaire de recherche.

2 Opérations de base

2.1 Rechercher un élément.

L'algorithme de recherche d'un élément dans un ABR est donné ci-dessous (algorithme 1).

```
Fonction Rechercher(clé x, ABR a) : booléen
begin
    si est–vide(a) alors
        return Faux
    sinon
        si val(a) == x alors
            return Vrai
        sinon
            si val(a) ≥ x alors
                return Rechercher(x,G(a))
            sinon
                return Rechercher(x,D(a))
end
```

Algorithme 1 : Rechercher un élément dans un ABR.

2.2 Ajouter un élément.

On l'ajoute sur une feuille (*i.e.* un arbre vide)... On remplace un arbre vide par un arbre contenant l'élément à ajouter et deux sous-arbres vides. Cette procédure est décrite par l'algorithme 2.

Exemple 2 *On applique, à partir d'un arbre vide, la procédure* `Ajouter` *aux clés suivantes :*

$$20, 15, 10, 35, 19, 13, 5, 3, 12, 7, 16, 40, 25, 38$$

```
Procédure Ajouter(clé x, ABR a)
begin
    si est–vide(a) alors
        a = Arbre(x, ArbreVide(), ArbreVide()) // remplace a par ...
    sinon
        si x ≤ val(a) alors
            Ajouter(x,G(a))
        sinon
            Ajouter(x,D(a))
end
```

Algorithme 2 : Ajouter un élément dans un ABR

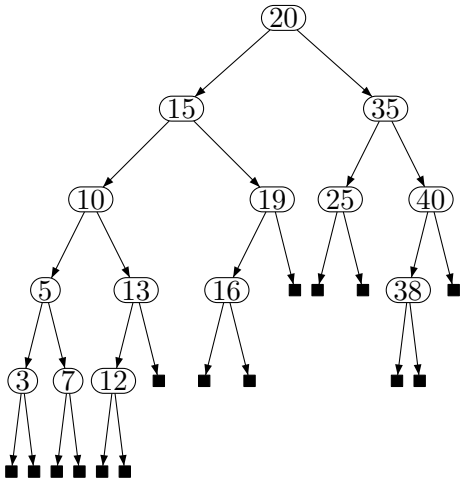

FIGURE 2 – Exemple d'ABR.

Cela donne l'ABR de la figure 2.

NB : *on aurait obtenu le même ABR avec, par exemple, la séquence suivante :*

$$20, 15, 10, 35, \boldsymbol{40}, 19, 13, 5, 3, 12, 7, 16, 25, 38$$

Cette remarque (deux séquences différentes d'ajouts peuvent conduire au même ABR) sera importante lors de l'analyse de complexité en moyenne.

NB : si on suppose qu'une clé ne peut apparaître qu'au plus une fois dans un ABR, il faut ajouter l'hypothèse "x n'est pas dans a" pour que l'algorithme soit correct.

Correction de l'algorithme. Lorsque l'on autorise plusieurs occurrences d'une même clé dans un ABR a, alors les clés contenues dans a ne constituent plus un ensemble mais un **ensemble avec répétition** appelé un **multi-ensemble**. Un multi-ensemble E de clés est défini par une fonction m_E de l'univers des clés (l'ensemble de toutes les clés possibles) dans $\mathbb{N}$: $m_E(x)$ représente le nombre d'occurrences de x dans E. On dit que "E contient x" ou "x appartient à E" (noté $x \in E$) lorsque $m_E(x) \geq 1$, on note $E + x$ l'ajout de x à E (c'est à dire le multi-ensemble E' défini par $m_{E'}$ telle que $m_{E'}(x) = m_E(x) + 1$ et $m_{E'}(y) = m_E(y)$ pour tout $y \neq x$) et $E - x$ la suppression de x de E (avec $m_{E'}(x) = \max(0, m_E(x) - 1)$,...)

Étant donné un ABR a, on note $S(a)$ le multi-ensemble des clés contenues dans a. On peut alors énoncer la correction des procédures `Ajouter` et `Rechercher` :

Proposition 1 (Correction)
Étant donné un ABR a contenant le multi-ensemble de clés S, et une clé x,
- *l'exécution de* `Ajouter`(x, a) *transforme a en un ABR contenant le multi-ensemble de clés $S + x$;*
- *l'exécution de* `Rechercher`(x, a) *retourne Vrai ssi x appartient à S, et Faux sinon.*

Preuve :

1. La preuve se fait par induction sur la taille de a. Si $|a| = 0$, l'arbre a est l'arbre vide : la procédure `Ajouter` transforme a en une feuille avec deux arbres vides. Si $|a| = n+1$, alors selon la valeur associée à la racine de a, on appelle récursivement la procédure `Ajouter` sur le sous-arbre gauche a_1 (contenant les clés S_1) ou le sous-arbre droit a_2 (contenant les clés S_2). Dans les deux cas, on applique l'hypothèse de récurrence et on obtient que le nouveau sous-arbre est un ABR et contient l'ensemble $S_i + x$, de plus l'arbre global vérifie toujours la propriété des ABR car x a été ajouté du bon coté...
2. Même démarche que ci-dessus : Si $|a| = 0$, la réponse est Faux et c'est correct. Si $|a| = n + 1$, alors si la racine de a contient x, la réponse de l'algorithme est bonne, sinon la structure d'ABR impose que x ne peut être que dans le sous-arbre gauche ou le sous-arbre droit selon le résultat du test "$x \leq$ `val`(a)", or l'hypothèse de récurrence nous assure que l'appel récursif sur le "bon" sous-arbre nous donne la bonne réponse.

□

2.3 Supprimer un élément.

Il s'agit à présent de supprimer une certaine clé x d'un ABR a. Pour cela, on va :

1. trouver sa place (*ie* son noeud interne) ;
2. le remplacer par le plus grand élément de son sous-arbre gauche ou par le plus petit élément de son sous-arbre droit. (Si le noeud à supprimer a un sous-arbre vide, la transformation est encore plus simple).

Cette procédure est décrite par les algorithmes 3 et 4.

On énonce la correction de la manière suivante :

Proposition 2 (Correction)
Étant donné un ABR a contenant le multi-ensemble de clés S, et une clé x, l'exécution de `Supprimer`(x, a) *transforme a en un ABR contenant le multi-ensemble de clés $S - x$*

Preuve : à faire en exercice ! □

2.4 Algorithmes itératifs

La structure d'arbre binaire se prête particulièrement bien aux algorithmes récursifs. On peut néanmojns utiliser des algorithmes itératifs très simples pour les opérations de base.

```
Procédure Supprimer(clé x, ABR a)
begin
    si ¬est–vide(a) alors
        si x < val(a) alors
            Supprimer(x,G(a))
        sinon
            si x > val(a) alors
                Supprimer(x,D(a))
            sinon
                si est–vide(G(a)) alors
                    a := D(a)
                sinon
                    si est–vide(D(a)) alors
                        a := G(a)
                    sinon
                        val(a) := Extraire-Max(G(a))
                        // voir algo. 4
end
```

Algorithme 3 : Supprimer un élément dans un ABR

```
Fonction Extraire-Max(ABR a) : clé
//On suppose que a est non vide !
begin
    si est–vide(D(a)) alors
        v := val(a)
        a := G(a)
        return v
    sinon
        return Extraire-Max(D(a))
end
```

Algorithme 4 : Extraire le max d'un ABR non vide.

```
Procédure Ajouter(clé x, ABR a)
begin
    r = a;
    tant que ¬est–vide(r) faire
        si x ≤ val(r) alors r := G(r);
        sinon r := D(r)
    r = Arbre(x, ArbreVide(), ArbreVide());
end
```

Algorithme 5 : Ajouter un élément dans un ABR (version itérative)

3 Tri par ABR

Lorsque l'on parcourt un arbre binaire pour opérer un certain traitement sur les noeuds, on distingue habituellement trois types de parcours : le **parcours préfixe** où on traite le

```
Fonction Rechercher(clé x, ABR a) : booléen
begin
    r = a;
    tant que ¬est–vide(r) ∧ val(r)!= x faire
        si x < val(r) alors r := G(r);
          sinon r := D(r)
    si est–vide(r) alors return Faux ;
      sinon return Vrai
end
```

Algorithme 6 : Rechercher un élément dans un ABR (version itérative).

noeud x dès qu'on le rencontre (avant de traiter ses sous-arbres), le **parcours infixe** où l'on traite de noeud x juste après le traitement de son sous-arbre gauche (et avant celui de son sous-arbre droit) et le **parcours postfixe** où l'on traite d'abord les sous-arbres avant le noeud x.

Pour afficher les clés dans un ordre croissant, il suffit de considérer un parcours infixe : On considère l'algorithme 7 de parcours d'un ABR avec affichage de la valeur des noeuds internes lors du "second" passage.

```
Procédure Parcours(ABR a)
begin
    si ¬est–vide(a) alors
        [premier passage]
        Parcours(G(a))
        [second passage : Afficher val(a)]
        Parcours(D(a))
        [troisième passage]
end
```

Algorithme 7 : Parcours d'un ABR

Étant donnée une séquence de nombres $x_1, \ldots, x_n$ à trier, un algorithme possible consiste à :

1. appeler `Ajouter`(a, x_i) pour tout i, en commençant avec a = l'arbre vide ;
2. appeler `Parcours`(a).

4 Analyse de complexité

Une bonne référence pour approfondir ces questions (notamment pour la complexité en moyenne) : "Introduction à l'analyse des algorithmes", R. Sedgewick, Ph. Flajolet, International Thomson Publishing.

La complexité des opérations sur les ABR sera mesurée en nombre de comparaisons de clés.

4.1 Définitions

Soit a un ABR.

On note :
- $\mathsf{N}_i(a)$ l'ensemble des noeuds *internes* de a (et on a $|\mathsf{N}_i(a)| = |a|$) ;
- $\mathsf{N}_{ext}(a)$ l'ensemble des noeuds *externes* (c.-à-d. correspondant à des arbres vides) de a ; et
- $\mathsf{N}(a)$ l'ensemble de tous les noeuds (internes ou externes) de a (on a bien sûr $\mathsf{N}(a) = \mathsf{N}_i(a) \cup \mathsf{N}_{ext}(a)$).

Nous avons la propriété suivante sur le lien entre le nombre de clés et le nombre de feuilles dans a :

Propriété 1 *Pour tout ABR a, nous avons : $|\mathsf{N}_{ext}(a)| = |a| + 1$.*

Preuve : Par induction sur $|a|$. A faire en exercice !

□

On définit aussi les notions suivantes :
- la profondeur d'un noeud s de a : $\mathsf{prof}_a(s) \stackrel{\text{def}}{=}$ la longueur du chemin reliant la racine de a à s.
  (si s est la racine, sa profondeur est 0 ; si s est un fils de la racine, sa profondeur est 1, *etc.*)
- la hauteur de a : $\mathsf{h}(a) \stackrel{\text{def}}{=} \max_{s \in \mathsf{N}(a)} \{\mathsf{prof}_a(s)\}$.

Exemple 3 *Si l'on considère l'ABR de la figure 2, alors sa taille est 14, la profondeur du noeud contenant 10 est 2, celle du noeud contenant 35 est 1, et la hauteur de l'arbre est 5.*

Notons que nous avons les deux propriétés suivantes :

Propriété 2
- $|a| = \begin{cases} 0 & \textit{si } \texttt{est\_vide}(a) \\ 1 + |\mathtt{G}(a)| + |\mathtt{D}(a)| & \textit{sinon} \end{cases}$
- $h(a) = \begin{cases} 0 & \textit{si } \texttt{est\_vide}(a) \\ 1 + \max(h(\mathtt{G}(a)), h(\mathtt{D}(a))) & \textit{sinon} \end{cases}$

On évalue la complexité des opérations `Rechercher`(x, a) et `Ajouter`(x, a). On va distinguer le coût des recherches *positives* (lorsque la clé x est trouvée dans l'arbre a), et le coût des recherches *négatives* (lorsque la clé x n'appartient pas à a).

On évalue la complexité des opérations en nombre de comparaisons de clés en fonction de la taille de a (notée n).

4.2 Complexité dans le pire cas

Recherche positive. Étant donné un ABR a contenant n clés, alors dans le pire cas, une recherche positive d'une clé x peut nécessiter n comparaisons : $n-1$ comparaisons "négatives" et 1 dernière comparaison "positive". Ce pire cas correspond au cas d'un arbre filiforme (où chaque noeud interne a au moins un fils qui est un arbre vide) de la forme décrite à la figure 3 (dans cet arbre, le pire cas est obtenu par la recherche de la clé 12).

Recherche négative. Le pire cas d'une recherche négative correspond aussi au cas d'ABR filiformes et son coût est alors de n comparaisons (négatives) : la clé x est comparée à toutes les clés de a avant de conclure à l'absence de x dans a. Dans le cas de l'ABR de la figure 3, un pire cas est obtenu, par exemple, avec la recherche de la clé 9.

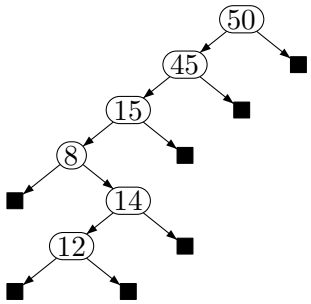

FIGURE 3 – Exemple d'ABR filiforme de taille 6.

Ajouter. Le pire cas pour la procédure `Ajouter` correspond aussi à un ABR filiforme pour lequel l'ajout de la clé x nécessite de passer par tous les noeuds internes pour trouver la feuille où insérer x. Le coût est alors de n comparaisons (négatives). Dans le cas de l'ABR de la figure 3, on obtient un pire cas avec, par exemple, `Ajouter(13,a)`.

Ces premiers résultats de complexité montrent que, dans le pire cas, un ABR n'est pas plus efficace qu'un simple tableau ou qu'une liste (qui correspond d'une certaine manière au cas des arbres filiformes).

La procédure `Supprimer` a aussi une complexité de n comparaisons dans le pire cas (toujours le cas des arbres filiformes).

La complexité de toutes ces opérations sur les ABR est directement liée à la **hauteur** de l'arbre sur lequel on applique l'opération (c'est à dire sur la longueur d'un plus long chemin menant de la racine à une feuille). Or cette hauteur, dans le pire cas, peut être linéaire (exactement n) dans la taille de l'arbre.

4.3 Complexité moyenne

Il s'agit maintenant d'évaluer la complexité **en moyenne** d'une recherche positive, d'une recherche négative et de l'opération `Ajouter`. Pour cela, nous allons nous intéresser à un ABR "**aléatoire** à n clés" et mesurer le coût **moyen** de ces différentes opérations sur cet arbre. Il faut préciser plusieurs points :

Qu'est-ce qu'un arbre aléatoire à n clés ? On suppose qu'un ABR de taille n est obtenu par l'application de la fonction `Ajouter` sur une suite de n clés (distinctes) c'est l'ordre de ces clés dans cette suite qui fixe l'ordre des appels à `Ajouter` et impose la forme de l'ABR. On supposera que toutes les $n!$ permutations sur $1 \cdots n$ sont équiprobables. Il y a donc $n!$ arbres à considérer et chacun a la probabilité $\frac{1}{n!}$.

Dans la suite, on note σ_n l'ensemble des $n!$ permutations sur $\{1, \ldots, n\}$. Et étant donnée p une permutation de $\sigma_n b$, on notera $a[p]$ l'ABR obtenu après les opérations `Ajouter`$(p(1))$, `Ajouter`$(p(2))$, ..., `Ajouter`$(p(n))$ à partir de l'arbre vide.

Notons que parmi ces $n!$ arbres, il y a beaucoup de doublons : deux séquences d'opérations `Ajouter` peuvent conduire au même ABR : nous l'avions déjà observé dans l'exemple 2, nous pouvons aussi le voir en considérons le cas avec $n = 3$:

Exemple 4 *L'ensemble σ_3 contient les permutations $\{1,2,3\}$, $\{1,3,2\}$, $\{2,1,3\}$, $\{2,3,1\}$, $\{3,1,2\}$ et $\{3,2,1\}$. Ces 6 permutations donnent les 6 ABR de la figure 4. On voit que les*

permutations $\{2,1,3\}$ *et* $\{2,3,1\}$ *donnent le même ABR.*

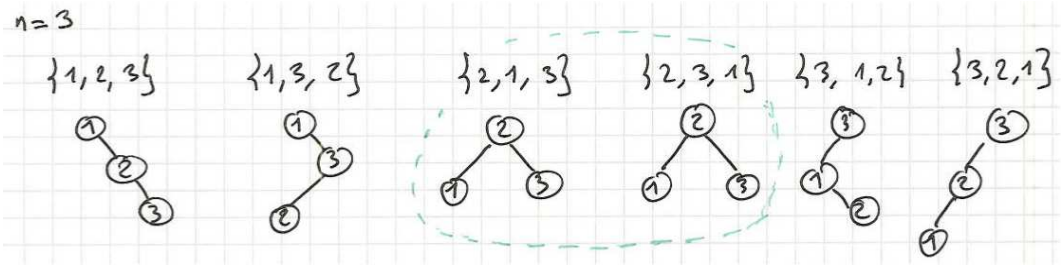

FIGURE 4 – Les 6 ABR pour σ_3

Cette remarque est importante car ce que nous allons calculer est bien une moyenne sur les $n!$ arbres et non sur l'ensemble (plus petit car sans doublon) des ABR à n clés.

Comment évaluer le "coût moyen" des opérations sur un arbre donné ? Étant donné un arbre a, le coût moyen d'une recherche positive est la somme des coûts de la recherche de chaque clé x présente dans a multipliés par la probabilité que la recherche porte sur x. On supposera que chaque clé est équiprobable, la probabilité est donc $\frac{1}{n}$. De plus, le coût de la recherche positive de x est $\mathsf{prof}_a(s_x)+1$ où s_x est le noeud interne contenant la clé x : il y a d'abord $\mathsf{prof}_a(s_x)$ tests négatifs avec les clés "au dessus" de x puis enfin un dernier test positif sur x. Le coût moyen d'une recherche positive dans a, noté $\mathsf{CR}_+(a)$, est donc : $\sum_{s\in \mathsf{N}_i(a)} \frac{1}{n}\mathsf{prof}_a(s)$.

Si l'on définit la **longueur de cheminement <u>interne</u> (LCI)** de a par :

$$\mathsf{LCI}(a) \stackrel{\text{def}}{=} \sum_{s\in \mathsf{N}_i(a)} \mathsf{prof}_a(s)$$

alors on obtient que le coût moyen d'une recherche positive dans a est donc : $\mathsf{CR}_+(a) = 1 + \frac{1}{n}\mathsf{LCI}(a)$

Une recherche négative est une recherche qui s'arrête sur une des $n+1$ feuilles (les arbres vides) de a (ce nombre $n+1$ vient de la propriété 1). On va supposer que ces $n+1$ cas sont équiprobables. On définit la **la longueur de cheminement <u>externe</u> (LCE)** de a défini par :

$$\mathsf{LCE}(a) \stackrel{\text{def}}{=} \sum_{s\in \mathsf{N}_{ext}(a)} \mathsf{prof}_a(s)$$

Le coût moyen d'une recherche négative pour a, noté $\mathsf{CR}_-(a)$, va donc correspondre à :

$$\mathsf{CR}_-(a) \;=\; \frac{1}{n+1}\mathsf{LCE}(a)$$

Enfin pour évaluer le coût moyen d'une opération `Ajouter`, il faut se rappeler que chacune de ces opérations commence par une recherche d'un arbre vide (une feuille de a) où on insère la nouvelle clé. Le coût de cette opération est donc le même que celui des recherches négatives. Là encore on supposera que les $n+1$ places où la nouvelle clé peut être insérée sont équiprobables, le coût moyen d'une opération `Ajouter`, noté $\mathsf{CA}(a)$, est donc $\frac{1}{n+1}\mathsf{LCE}(a)$.

Enfin on peut aussi considérer le **coût total** de construction d'un ABR a, c'est-à-dire la somme des coûts de toutes les opérations `Ajouter` qui ont été réalisées pour le construire. Ce coût, noté $\mathsf{Ct}(a)$ correspond à la longueur de cheminement interne de a : en effet, pour chaque clé x, son insertion à demander de réaliser $\mathsf{prof}_s(s_x)$ comparaisons... On a donc : $\mathsf{Ct}(a) = \mathsf{LCI}(a)$.

De toutes ces remarques, il ressort que les valeurs $\mathsf{LCI}(a)$ et $\mathsf{LCE}(a)$ sont fondamentales pour évaluer la complexité en moyenne des opérations sur les ABR. Elles interviennent dans toutes les mesures considérées. Et il s'agit à présent de les calculer pour **un ABR aléatoire à n clés**. Pour cela, nous allons d'abord montrer le lien entre $\mathsf{LCE}(a)$ et $\mathsf{LCI}(a)$.

On a la propriété suivante :

Propriété 3 *Pour tout ABR non vide a, nous avons :*
- $LCI(a) = LCI(\mathtt{G}(a)) + LCI(\mathtt{D}(a)) + |a| - 1$
- $LCE(a) = LCE(\mathtt{G}(a)) + LCE(\mathtt{D}(a)) + |a| + 1$

Preuve : On a bien $\mathsf{LCI}(a) = \mathsf{LCI}(\mathtt{G}(a)) + \mathsf{LCI}(\mathtt{D}(a)) + |a| - 1$ car pour chaque noeud interne de a à l'exception de la racine, il faut ajouter 1 à sa profondeur dans le sous-arbre gauche ou droit pour retrouver sa hauteur dans a.
Le même raisonnement s'applique pour $\mathsf{LCE}(a)$ mais cette fois il faut ajouter 1 pour tous les noeuds externes (il y en a $|a| + 1$). □

On en déduit :

Propriété 4 *Pour tout ABR a, on a :* $LCE(a) = LCI(a) + 2 \cdot |a|$

Preuve : A faire en exercice !

□

Ce dernier résultat permet donc de déduire LCE de LCI et vice-versa. Il nous reste donc à calculer l'une de ces deux quantités pour obtenir la complexité en moyenne de nos opérations.

Pour cela, nous allons montrer deux méthodes différentes : la première va partir de la définition de la longueur de cheminement externe d'un ABR aléatoire (la moyenne pondérée des LCE sur les $n!$ ABR à n clés), la seconde va procéder autrement en raisonnant directement sur le **coût total des opérations `Ajouter` réalisées pour le construire** (qui, comme nous avons noté précédemment, correspond à la LCI).

4.3.1 Première méthode

Chaque ABR avec n clés a été obtenu par l'application successive de n opérations `Ajouter` sur ces clés. Il y a $n!$ permutations possibles de ces n clés, cela donne $n!$ arbres binaires.

La longueur de cheminement externe (LCE) d'un ABR aléatoire à n éléments, notée C_n, est définie comme la moyenne des LCE des $n!$ ABR obtenus par toutes les permutations de σ_n :

$$C_n \stackrel{\text{def}}{=} \frac{1}{n!} \sum_{p \in \sigma_n} \mathsf{LCE}(a[p])$$

où $a[p]$ désigne l'ABR obtenu après les opérations `Ajouter`$(p(1))$, `Ajouter`$(p(2))$, ..., et `Ajouter`$(p(n))$ sur l'arbre vide.

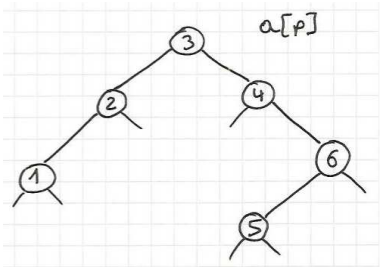

FIGURE 5 – $a[p]$ avec $p = \{3, 4, 2, 6, 5, 1\}$

Exemple 5 *Si p est la permutation $\{3, 4, 2, 6, 5, 1\}$ de σ_6, alors $a[p]$ est l'ABR décrit à la figure 5. Notons que par exemple, $a[p']$ avec $p' = \{3, 2, 4, 6, 5, 1\}$ est identique à $a[p]$.*

On peut regrouper les permutations qui ont le même premier élément :

$$C_n \stackrel{\text{def}}{=} \frac{1}{n!} \sum_{k=1}^{n} \sum_{p \in \sigma_n \ tq \ p(1)=k} \mathsf{LCE}(a[p])$$

Le choix du premier élément est important car c'est cet élément qui sera mis à la racine de l'ABR : toutes les permutations p de σ_n ayant $p(1) = k$ donneront un ABR avec la clé k à la racine.

Une fois le premier élément k fixé, c'est le "reste" de la permutation (c.-à-d. $p' = \{p(2), p(3), \ldots, p(n)\}$) qui va définir l'ABR correspondant à p. Or la construction du sous-arbre gauche ne dépend que de l'ordre des éléments inférieurs à k dans p'. Et celle du sous-arbre droit ne dépend que de l'ordre des éléments supérieurs à k. Si l'on voit la permutation p' comme étant obtenue par le "mélange" d'une permutation p_1 sur $\{1, \ldots, k-1\}$ et d'une permutation p_2 sur $\{k+1, \ldots, n\}$, alors la manière dont p_1 et p_2 sont "mélangées pour former p' n'influe pas sur la structure de l'ABR final.

Exemple 6 *Si l'on reprend la permutation $p = \{3, 4, 2, 6, 5, 1\}$ de σ_6. Ici on a $k = 3$. p' est obtenue par un mélange de $p_1 = \{2, 1\}$ et $p_2 = \{4, 6, 5\}$. Mais un autre mélange de p_1 et p_2, par exemple $\{3, 2, 4, 1, 6, 5\}$ aurait donné exactement le même ABR que celui dessiné ci-dessus pour p.*

On note à présent $a[k, p_1, p_2]$ l'ABR ayant la clé k à la racine et dont le sous-arbre gauche est construit selon la permutation p_1 (sur les éléments inférieurs à k) et dont le sous-arbre droit est construit selon p_2 (sur les éléments supérieurs à k). Alors on a :

$$C_n = \frac{1}{n!} \sum_{k=1}^{n} \Big(\sum_{p_1 \in \sigma_{k-1}} \sum_{p_2 \in \sigma'_{n-k}} \text{"nb de mélanges possibles pour } p_1 \text{ et } p_2\text{"} \cdot \mathsf{LCE}(a[k, p_1, p_2])$$

Ici on a noté σ'_{n-k} l'ensemble des permutations sur $\{k+1, \ldots, n\}$.

Exercice : Expliquer l'équation ci-dessus.

Il reste maintenant à évaluer le nombre de mélanges possibles pour les deux permutations p_1 et p_2. Ce nombre dépend de k. On peut voir ce nombre Nb_k comme le nombre de sous-ensembles de taille $k-1$ dans un ensemble de taille $n-1$ ou, de manière équivalente, comme

le nombre de sous-ensembles de taille $n-k$ dans un ensemble de taille $n-1$:

$$\mathsf{Nb}_k \stackrel{\text{def}}{=} \binom{n-1}{k-1} = \binom{n-1}{n-k} = \frac{(n-1)!}{(n-k)!(k-1)!}$$

Exercice : Expliquer la valeur de Nb_k.

On en déduit :

$$C_n = \frac{1}{n!}\sum_{k=1}^{n}\left(\sum_{p_1\in\sigma_{k-1}}\sum_{p_2\in\sigma'_{n-k}}\binom{n-1}{k-1}\cdot\mathsf{LCE}(a[k,p_1,p_2])\right)$$

Et donc :

$$C_n = \frac{1}{n!}\sum_{k=1}^{n}\binom{n-1}{k-1}\cdot\left(\sum_{p_1\in\sigma_{k-1}}\sum_{p_2\in\sigma'_{n-k}}\mathsf{LCE}(a[k,p_1,p_2])\right)$$

D'après la propriété 3, on a $\mathsf{LCE}(a) = \mathsf{LCE}(\mathtt{G}(a)) + \mathsf{LCE}(\mathtt{D}(a)) + |a| + 1$. De plus, $\mathsf{LCE}(\mathtt{G}(a[k,p_1,p_2])) = \mathsf{LCE}(a[p_1])$ car p_2 n'intervient pas dans la construction du sous-arbre gauche... Et de même, on a $\mathsf{LCE}(\mathtt{D}(a[k,p_1,p_2])) = \mathsf{LCE}(a[p_2])$. On en déduit :

$$C_n = \frac{1}{n!}\sum_{k=1}^{n}\binom{n-1}{k-1}\cdot\left(\sum_{p_1\in\sigma_{k-1}}\sum_{p_2\in\sigma'_{n-k}}\mathsf{LCE}(a[p_1]+\mathsf{LCE}(a[p_2])+n+1)\right)$$

Comme le nombre de permutations de σ_{k-1} est $(k-1)!$ et celui de σ'_{n-k} est $(n-k)!$, on a :

$$C_n = \frac{1}{n!}\sum_{k=1}^{n}\binom{n-1}{k-1}\cdot\left((n-k)!\cdot\sum_{p_1\in\sigma_{k-1}}\mathsf{LCE}(a[p_1] + (k-1)!\cdot\sum_{p_2\in\sigma'_{n-k}}\mathsf{LCE}(a[p_2]) + (n-k)!\cdot(k-1)!\cdot(n+1)\right)$$

On a aussi $\sum_{p_2\in\sigma'_{n-k}}\mathsf{LCE}(a[p_2]) = \sum_{p_2\in\sigma_{n-k}}\mathsf{LCE}(a[p_2])$ car seul compte le nombre de clés (et non leur valeur). D'où :

$$C_n = \frac{1}{n!}\sum_{k=1}^{n}\binom{n-1}{k-1}\cdot\left((n-k)!\cdot\sum_{p_1\in\sigma_{k-1}}\mathsf{LCE}(a[p_1] + (k-1)!\cdot\sum_{p_2\in\sigma_{n-k}}\mathsf{LCE}(a[p_2]) + (n-k)!\cdot(k-1)!\cdot(n+1)\right)$$

Et donc :

$$C_n = \frac{1}{n!}\sum_{k=1}^{n}\frac{(n-1)!}{(k-1)!(n-k)!}\cdot\left((n-k)!\cdot\sum_{p_1\in\sigma_{k-1}}\mathsf{LCE}(a[p_1] + (k-1)!\cdot\sum_{p_2\in\sigma_{n-k}}\mathsf{LCE}(a[p_2]) + (n-k)!\cdot(k-1)!\cdot(n+1)\right)$$

Et finalement :

$$C_n \;=\; \frac{1}{n}\sum_{k=1}^{n}\left(\frac{1}{(k-1)!}\cdot\sum_{p_1\in\sigma_{k-1}}\mathsf{LCE}(a[p_1]\;+\;\frac{1}{(n-k)!}\cdot\sum_{p_2\in\sigma_{n-k}}\mathsf{LCE}(a[p_2])\;+\;(n+1)\right)$$

Et enfin :

$$C_n \;=\; \frac{1}{n}\sum_{k=1}^{n}\Big(C_{k-1}+C_{n-k}+n+1\Big) \;=\; n+1+\frac{1}{n}\sum_{k=1}^{n}\Big(C_{k-1}+C_{n-k}\Big)$$

Cette formule correspond à la formule de complexité en moyenne du Tri rapide (Quicksort).

4.3.2 Seconde méthode

On montre à présent une autre manière (plus directe) pour trouver la récurrence précédente (développée dans "Introduction à l'analyse des algorithmes", R. Sedgewick, Ph. Flajolet, International Thomson Publishing).

Le point de départ de cette approche est de remarquer que la longueur de cheminement interne d'un ABR a correspond au **coût total de la construction de cet ABR** (c'est-à-dire le coût de toutes les opérations `Ajouter` effectuées pour construire l'arbre) : chaque appel de `Ajouter` s'est arrêté sur un de ces noeuds et leur profondeur n'a pas changé depuis.

C'est cette quantité, le coût total de construction, que nous allons évaluer un ABR aléatoire avec n clés. Nous la noterons Ct_n dans la suite.

Considérons une permutation $\{x_1,\ldots,x_n\}$. On commence par appliquer la procédure `Ajouter` à l'arbre vide et à la première clé x_1 de la suite. La clé x_1 est alors placée à la racine de l'arbre. Ensuite, les deux sous-arbres gauche et droit se construisent **indépendamment** : le sous-arbre gauche ne dépend que des clés inférieures à x_1, et le sous-arbre droit ne dépend que des clés supérieures à x_1.

Comme nous l'avons déjà expliqué dans le cas précédent, après x_1, la suite $(x_2,\ldots,x_n)$ de la permutation initiale correspond au "mélange" de deux permutations : la première porte sur les clés inférieures à x_1, et la seconde sur les clés supérieures à x_1. La manière de mélanger ces deux permutations n'influe pas sur la forme finale de l'ABR : seule compte l'ordre des clés apparaissant dans les deux sous-suites. C'est ce qui explique que l'on obtienne le même ABR après "4, 1, 2, 5, 6" ou "4, 1, 5, 2, 6" : les deux sous-suites "1, 2" et "5, 6" sont identiques. De plus, les permutations des clés inférieures à x_1, comme celles pour les clés supérieures à x_1, sont équiprobables.

La probabilité que la première clé x_1 soit la k^{eme} plus petite clé de l'ensemble est $\frac{1}{n}$. Dans ce cas, le sous-arbre gauche sera construit par l'insertion de $k-1$ clés (inférieures à x_1), et le sous-arbre droit sera construit par l'insertion de $n-k$ clés.

De ces remarques et de la formule de la propriété 3 (établie pour la LCI mais évidemment valable pour le coût total de construction d'un ABR), on déduit la formule de récurrence suivante qui donne le coût total moyen Ct_n de la **construction d'un ABR par insertion de n clés dans un ordre aléatoire** (*i.e.* sa "LCI moyenne") :

$$\mathrm{Ct}_n \;\stackrel{\text{def}}{=}\; \begin{cases} 0 & \text{si } n=0 \\ \dfrac{1}{n}\displaystyle\sum_{1\le k\le n}\Big(\mathrm{Ct}_{k-1}+\mathrm{Ct}_{n-k}+(n-1)\Big) & \text{sinon} \end{cases}$$

Lorsque $n > 0$, on a donc : $\mathrm{Ct}_n = (n-1) + \frac{1}{n} \sum_{1 \leq k \leq n} \Big(\mathrm{Ct}_{k-1} + \mathrm{Ct}_{n-k} \Big)$.

De la même manière, on peut exprimer sous cette forme la longueur de cheminement externe d'un ABR aléatoire C_n (ou même appliquer la propriété 4 à la définition de Ct_n qui correspond à la LCI d'un ABR aléatoire) et on obtient alors :

$$C_n \stackrel{\text{def}}{=} \begin{cases} 0 & \text{si } n = 0 \\ \frac{1}{n} \sum_{1 \leq k \leq n} \Big(C_{k-1} + C_{n-k} + (n+1) \Big) & \text{sinon} \end{cases}$$

Et donc : $C_n = (n+1) + \frac{1}{n} \sum_{1 \leq k \leq n} \Big(C_{k-1} + C_{n-k} \Big)$ si $n > 0$. On retrouve donc la même récurrence qu'avec la première méthode.

4.3.3 Résolution de la récurrence

On peut alors résoudre la récurrence de la manière suivante (et vous avez vu d'autres manières dans le cours de maths discrètes) :

$$\begin{aligned} C_n &= (n+1) + \frac{2}{n} \sum_{0 \leq k \leq n-1} C_k \\ C_n &= (n+1) + \frac{2}{n} \sum_{1 \leq k \leq n-1} C_k \end{aligned}$$

$$\begin{aligned} n \cdot C_n &= n(n+1) + 2 \sum_{1 \leq k \leq n-1} C_k \\ \text{d'où} \quad (n-1) \cdot C_{n-1} &= (n-1)n + 2 \sum_{1 \leq k \leq n-2} C_k \\ \text{donc} \quad n \cdot C_n - (n-1) \cdot C_{n-1} &= n(n+1) - n(n-1) + 2 \cdot C_{n-1} \end{aligned}$$

$$\begin{aligned} \text{donc} \quad n \cdot C_n &= (n+1)C_{n-1} + 2n \\ \frac{C_n}{n+1} &= \frac{C_{n-1}}{n} + \frac{2}{n+1} \\ \frac{C_n}{n+1} &= \frac{2}{n+1} + \frac{2}{n} + \ldots + \frac{2}{3} + \frac{2}{2} + 0 \\ \frac{C_n}{n+1} &= 2H_{n+1} - 2 \\ C_n &= 2(n+1)(H_{n+1} - 1) \end{aligned}$$

Où H_n est le n-ème nombre harmonique, défini par : $H_n \stackrel{\text{def}}{=} \sum_{i=1}^{n} \frac{1}{i}$, et on rappelle que :

$$\log(n+1) < H_n < \log(n) + 1$$

La longueur de cheminement externe d'un ABR obtenu par n insertions aléatoires est donc : $2(n+1)(H_{n+1} - 1)$.

Grâce à la propriété 4, on en déduit que la longueur de cheminement interne moyenne est : $2(n+1)(H_{n+1} - 1) - 2n$. On en déduit donc immédiatement le coût total de construction d'un ABR aléatoire car cette mesure est précisément sa LCI :

Proposition 3 *Le nombre moyen de comparaisons de clés effectuées pour construire un ABR en insérant n clés distinctes dans un ordre aléatoire à partir d'un ABR vide est :*

$$2(n+1)(H_{n+1}-1)-2n$$

Ce nombre est donc en $O(n \cdot \log(n))$.

4.3.4 Analyse de complexité

On en déduit à présent les différentes complexités en moyenne. Pour la recherche positive, on suppose que les n clés de l'ABR sont équiprobables :

Propriété 5 *Le coût moyen d'une recherche positive dans un ABR obtenu par n insertions aléatoires est :*

$$2H_n - 3 + 2\frac{H_n}{n}$$

Ce coût est donc en $O(\log(n))$.

Preuve : La recherche positive d'une clé coûte exactement le même nombre de comparaisons que son ajout PLUS 1. On en déduit donc que ce coût moyen d'une recherche positive dans un ABR aléatoire est :

$$K_n \stackrel{\text{def}}{=} \frac{1}{n}\Big(\text{"LCI moyenne"} + n\Big)$$

C'est à dire :

$$K_n = \frac{1}{n}\Big(2(n+1)(H_{n+1}-1) - 2n + n\Big)$$

Exercice : à développer...

□

Si l'on suppose que dans le cas d'une recherche négative, toutes les feuilles (c'est à dire les arbres vides) ont la même probabilité d'être atteintes, on a :

Propriété 6 *Le coût moyen d'une recherche négative dans un ABR obtenu par n insertions aléatoires est :*

$$2(H_{n+1}-1)$$

Ce coût est donc en $O(\log(n))$.

Preuve : Le coût moyen d'une recherche négative est la longueur de cheminement externe divisée par le nombre de feuilles $(n+1)$. On en déduit que ce coût moyen dans le cas d'un arbre aléatoire est la LCE moyenne divisée par $n+1$. □

De même, si l'on suppose que dans le cas d'un ajout, toutes les feuilles (c'est à dire les arbres vides) ont la même probabilité d'être atteintes (et associées à la nouvelle clé x), on a :

Propriété 7 *Le coût moyen d'un ajout dans un ABR aléatoire de taille n est :*

$$2(H_{n+1}-1)$$

Ce coût est donc en $O(\log(n))$.

Conclusion sur les ABR : la complexité en moyenne des ABR est donc bonne ! Par exemple, trier n clés selon la méthode donnée à la section 3 aura une complexité en moyenne en $O(n \cdot \log(n))$: c'est en effet la complexité en moyenne pour la construction totale de l'arbre et la complexité du parcours est toujours en $O(n)$. C'est donc un bon algorithme de tri.

De même, l'utilisation des ABR pour gérer des dictionnaires (stockage d'information, puis recherche) peut se faire de manière assez efficace en moyenne.

5 Arbres binaires de recherche équilibrés

L'objectif est d'assurer une complexité en $O(\log(n))$ même dans le pire cas pour les opérations d'ajout, de recherche et de suppression.

L'idée est d'utiliser des arbres **équilibrés** et de les maintenir équilibrés après les opérations d'ajout et de suppression. Plusieurs notions d'arbres équilibrés existent : les AVL, les arbres 2-3-4, les arbres rouges et noirs,... **Ils ont tous la propriété d'avoir une hauteur en $O(\log(n))$.**

Ici on s'intéresse aux arbres AVL (proposés par G. M. Adelson-Velskii et E. M. Landis) définis dans les années 60. Il s'agit d'arbres maintenus équilibrés à l'aide de rotations locales. Une présentation détaillée des AVL se trouve dans "Types de données et algorithmes" de Marie-Claude Gaudel , Christine Froidevaux et Michèle Soria (Ediscience International).

5.1 Définitions

Un AVL est un arbre binaire de recherche H-équilibré :

Définition 6 *Un arbre binaire a est H-équilibré ssi en tout noeud de a, les hauteurs des sous-arbres gauche et droit diffèrent au plus de 1.*

On définit la fonction suivante :

$$\mathsf{deseq}(a) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{si } \mathit{estvide}(a) \\ \mathsf{h}(\mathtt{G}(a)) - h(\mathtt{D}(a)) & \text{sinon} \end{cases}$$

Un AVL a est donc un ABR tel que tout noeud s de a vérifie : $\mathsf{deseq}(s) \in \{-1, 0, 1\}$.

Exemple 7 *La figure 6 montre un arbre binaire où la fonction de déséquilibre est donnée pour chaque noeud. Cet arbre est bien H-équilibré.*

Nous avons la propriété suivante qui motive l'utilisation de cette notion d'arbre équilibré :

Propriété 8 *Tout arbre H-équilibré a de taille n vérifie :*

$$\log_2(n+1) \ \leq \ \mathsf{h}(a) \ \leq \ 1.44 \cdot \log_2(n+1)$$

Preuve : La hauteur d'un arbre H-équilibré est supérieure ou égale à celle d'un arbre complet (c'est-à-dire où tous les niveaux sont remplis). Or pour un arbre complet, on a : $n = 1 + 2 + 2^2 + \ldots + 2^{h-1} = 2^h - 1$. On en déduit $\log_2(n+1) \leq h$.

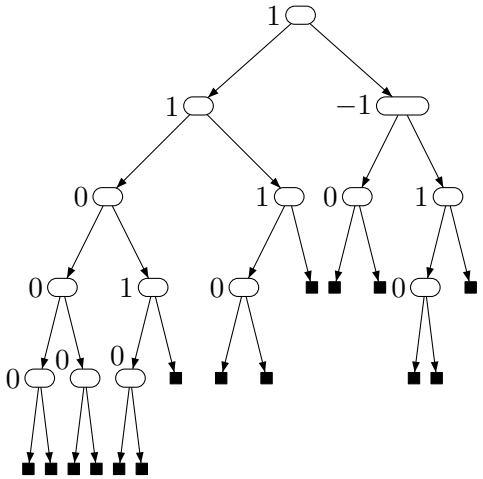

FIGURE 6 – Exemple d'arbre binaire avec les valeurs de $\mathsf{deseq}(-)$.

Étant donné h, les arbres H-équilibrés ayant le moins de noeuds internes et une hauteur h, sont ceux qui ont une fonction de déséquilibre valant 1 ou -1 partout (sinon on peut toujours enlever des noeuds internes).
Avec un tel arbre, après toute suppression d'une feuille, le nouvel arbre a' est soit non H-équilibré, soit on a $\mathsf{h}(a') = h - 1$.
Calculons le nombre de noeuds internes $Nb(h)$ dans ce genre d'arbre.

Comme la fonction de déséquilibre vaut 1 ou -1 à la racine de a, on a soit $\mathsf{h}(\mathtt{G}(a)) = h-1$ et $\mathsf{h}(\mathtt{D}(a)) = h - 2$, soit l'inverse : $\mathsf{h}(\mathtt{G}(a)) = h - 2$ et $\mathsf{h}(\mathtt{D}(a)) = h - 1$.

Dans les deux cas, on a : $Nb(h) = 1+Nb(h-1)+Nb(h-2)$, avec $Nb(0) = 0$ et $Nb(1) = 1$.

On peut utiliser la suite de Fibonacci : $F_1 = F_2 = 1$ et pour tout $k > 1$, on a $F_k = F_{k-1} + F_{k-2}$.

On a le résultat suivant : $F_k = \dfrac{1}{\sqrt{5}}(\phi^k - \bar{\phi}^k)$ avec $\phi = \dfrac{1+\sqrt{5}}{2}$ et $\bar{\phi} = \dfrac{1-\sqrt{5}}{2}$. Et asymptotiquement, on a : $F_k \sim \frac{1}{\sqrt{5}}\phi^k$.

On montre facilement (par exemple, par induction sur h) que $Nb(h) = F_{h+2} - 1$.

On en déduit qu'asymptotiquement, on a : $Nb(h) \sim \dfrac{1}{\sqrt{5}}\phi^{h+2} \; - \; 1$.

Donc tout arbre H-équilibré (de hauteur h) a un nombre de noeuds internes n supérieur ou égal à $Nb(h)$. On en déduit une majoration de h en fonction de n :

$$h \;\leq\; \frac{1}{\log_2(\phi)} \cdot \log_2(n+1)$$

On peut conclure $h \leq 1.44 \cdot \log_2(n+1)$. □

Les rotations. On définit plusieurs types de rotations (voir les figures 7, 8, 9 et 10) qui vont nous permettre de manipuler les arbres afin de garantir qu'ils restent bien des AVL après l'ajout de nouvelles clés.

On suppose $x_1 \leq x_2$. Le noeud grisé est celui sur lequel on applique la rotation.

Nous avons la propriété suivante :

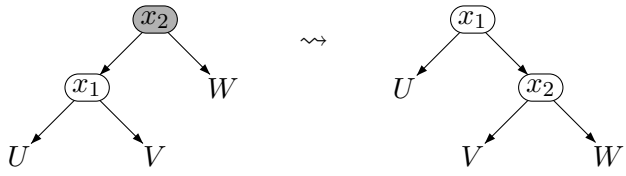

FIGURE 7 – Rotation droite

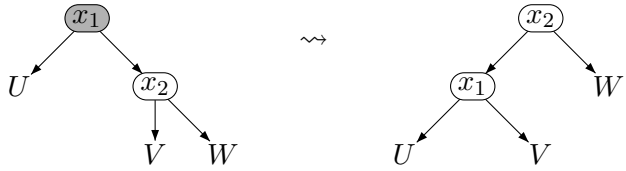

FIGURE 8 – Rotation gauche

Propriété 9 *Les opérations de rotation conservent la propriété d'arbre binaire de recherche.*

Preuve : A faire en exercice ... □

En conséquence : après ces transformations, le parcours de l'ABR utilisé pour le tri (pour obtenir les clés dans l'ordre croissant) donne toujours le même résultat.

5.2 Ajout dans un AVL

L'opération `Rechercher` est identique à celle des ABR classiques. Mais les opérations `Ajouter` et `Supprimer` doivent être modifiées substantiellement. Dans la suite, on notera `Ajouter-abr` la procédure `Ajouter` sur les ABR pour la distinguer de celle utilisée pour les AVL.

Exemple 8 *Ajouter les clés :* $12, 3, 2, 5, 4, 7, 9, 11, 14$ *(voir la figure 11).*

Pour décrire la procédure d'ajout dans les AVL, on part de celle dans les ABR classiques. On va voir comment :

- détecter les éventuels problèmes (le déséquilibre d'un noeud passe de -1, 0 ou 1 à -2 ou 2) ;
- réparer (par application d'une rotation) l'arbre **localement**.

Détection (et localisation) des éventuels problèmes :

Propriété 10 *Soit a un AVL et x une nouvelle clé. L'appel de* `Ajouter-abr`(x, a) *conduit à placer x sur un noeud externe t de a. On note a' le nouvel arbre.*

Soit π le chemin reliant la racine r de l'arbre a à t.

Alors :

- *si π ne contient que des sommets s tels que $\mathsf{deseq}_a(s) = 0$, alors a' est toujours un AVL, et $\mathsf{h}(a') = \mathsf{h}(a) + 1$;*
- *sinon, soit ρ le plus petit suffixe de π commençant en un sommet ayant un déséquilibre dans $\{-1, 1\}$ dans a. Supposons $\rho \stackrel{def}{=} t_0 \to t_1 \to \ldots \to t_k = t$, alors on a les propriétés suivantes :*
  - $\mathsf{deseq}_{a'}(t_0) \in \{-2, 0, 2\}$;
  - $\mathsf{deseq}_{a'}(t_i) \in \{-1, 1\}$ *pour* $i = 1, \ldots, k-1$;

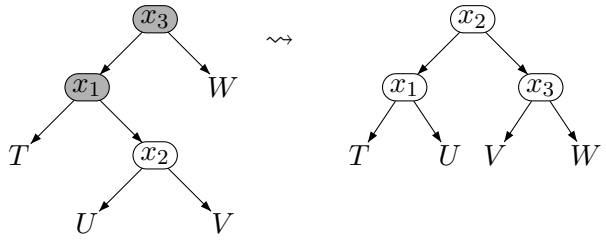

FIGURE 9 – Rotation gauche-droite : g sur x_1 puis d sur x_3

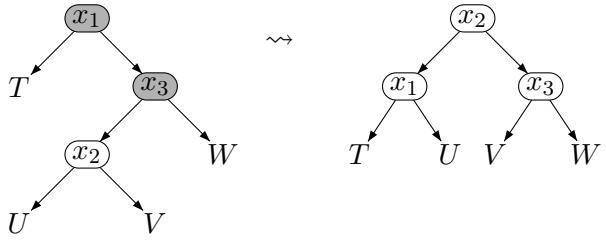

FIGURE 10 – Rotation droite-gauche : d sur x_3 puis g sur x_1

- $\mathsf{h}_{a'}(t_0) = \begin{cases} \mathsf{h}_a(t_0) & si \quad \mathsf{deseq}_{a'}(t_0) = 0 \\ \mathsf{h}_a(t_0) + 1 & sinon \end{cases}$
- $\mathsf{h}_{a'}(t_i) = h_a(t_i) + 1$ *pour* $i = 1, \ldots, k$.

Preuve : Supposons que tous les sommets de $\pi = r_0 \to \ldots \to r_k = t$ ont un déséquilibre nul dans a (et donc r_0 est la racine r de a). Le cas où le chemin π est constitué d'un unique sommet (a est vide) est direct : le nouvel ABR est bien un AVL. Autrement la transformation de t en un noeud interne contenant x, va déséquilibrer r_{k-1} puisqu'il était forcément muni de deux sous-arbres vides dans a : $\mathsf{deseq}_{a'}(r_{k-1}) \in \{-1, 1\}$. De même, on a clairement $\mathsf{h}_{a'}(r_{k-1}) = h_a(r_{k-1}) + 1 = 2$. Le déséquilibre va donc se propager à r_{k-2}, etc. Finalement, on aura $\mathsf{deseq}_{a'}(r_i) \in \{-1, 1\}$ et a' restera un AVL. Et on a bien $\mathsf{h}(a') = \mathsf{h}(a) + 1$.

Maintenant supposons que le suffixe ρ existe (voir la figure 12). On a $\rho \stackrel{\text{def}}{=} t_0 \to t_1 \to \ldots \to t_k = t$ avec $\mathsf{deseq}_a(t_0) \in \{-1, 1\}$ et $\mathsf{deseq}_a(t_i) = 0$ pour $i = 1, \ldots, k$. Le sous-arbre de a ayant pour racine le noeud (externe) t a une hauteur nulle. Il est remplacé dans a' par un arbre de hauteur 1. Si son père t_{k-1} avait un déséquilibre nul dans a, alors il obtient un déséquilibre de 1 ou -1 dans a' selon que t soit son fils gauche ou son fils droit.
De même, on a $\mathsf{h}_{a'}(t_{k-1}) = \mathsf{h}_a(t_{k-1}) + 1$ et cette croissance va alors influer sur le déséquilibre de son père t_{k-2}, etc. Donc tous les noeuds de ρ ayant un déséquilibre nul (à l'exception de $t = t_k$) vont avoir un déséquilibre de 1 ou -1 dans a'. Et leur hauteur (c'est-à-dire celle de leur sous-arbre) va augmenter de 1.
Finalement on arrive à t_0, on distingue plusieurs cas :

- si $\mathsf{deseq}_a(t_0) = 1$ et si t_1 est le fils gauche de t_0, alors on a $\mathsf{deseq}_{a'}(t_0) = 0$: l'augmentation de la hauteur du sous-arbre gauche de t_0 rééquilibre l'arbre de racine t_0 (sans augmenter sa hauteur) ;
- si $\mathsf{deseq}_a(t_0) = -1$ et si t_1 est le fils droit de t_0, alors on a $\mathsf{deseq}_{a'}(t_0) = 0$: cas symétrique...
- si $\mathsf{deseq}_a(t_0) = -1$ et si t_1 est le fils gauche de t_0, alors on a $\mathsf{deseq}_{a'}(t_0) = -2$, et ;

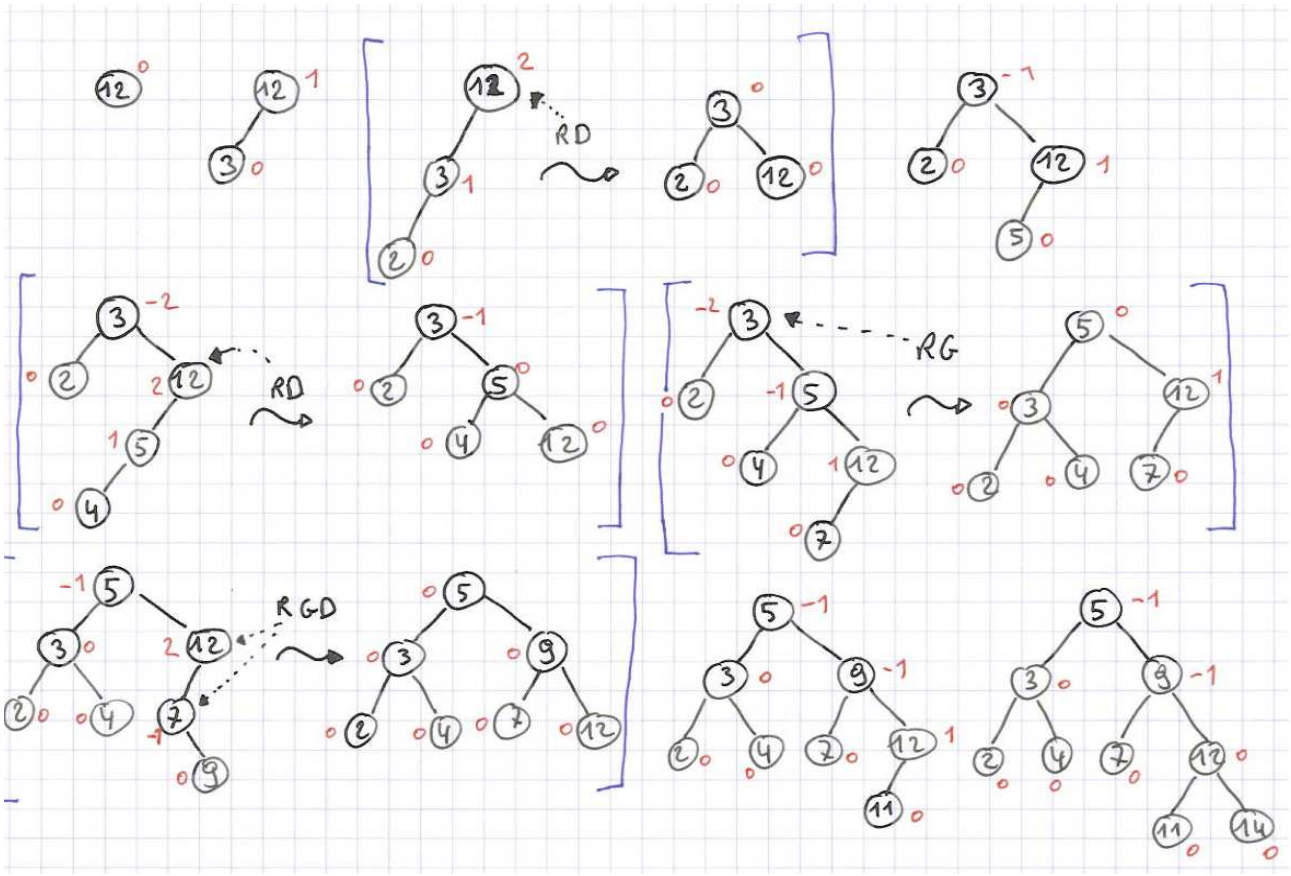

FIGURE 11 – Ajout de $12, 3, 2, 5, 4, 7, 9, 11, 14$

– si $\mathsf{deseq}_a(t_0) = 1$ et si t_1 est le fils droit de t_0, alors on a $\mathsf{deseq}_{a'}(t_0) = 2$.

La hauteur du sous-arbre de a' ayant t_0 pour racine sera donc $\mathsf{h}_a(t_0)$ si le nouveau déséquilibre de t_0 est 0, ou alors $\mathsf{h}_a(t_0) + 1$ sinon.

□

Lorsqu'il y a déséquilibre après un ajout, le sommet t_0 décrit ci-dessus est donc le sommet le plus profond ayant un déséquilibre $\{-2, 2\}$ dans a'.

Avant de voir comment on peut réparer l'arbre précédent, on considère le lemme suivant qui nous servira dans la suite :

Lemme 1 *Soit a un arbre H-équilibré. Soit b un sous-arbre de a.*
Soit b' un arbre H-équilibré de même hauteur que b.
Alors l'arbre a' constitué de l'arbre a dans lequel b est remplacé par b', est H-équilibré.

Preuve :

On est dans la situation décrite par la figure 13.

Considérons un sommet x de a'. On a : $\mathsf{deseq}_{a'}(x) = \mathsf{h}(\mathtt{G}_{a'}(x)) - \mathsf{h}(\mathtt{D}_{a'}(x))$. Si x est un noeud de b', alors $\mathsf{deseq}_{a'}(x) = \mathsf{deseq}_{b'}(x) \in \{-1, 0, 1\}$ car b' est H-équilibré.

Si le sous-arbre de racine x est disjoint de b', alors $\mathsf{deseq}_{a'}(x) \in \{-1, 0, 1\}$ car a est H-équilibré.

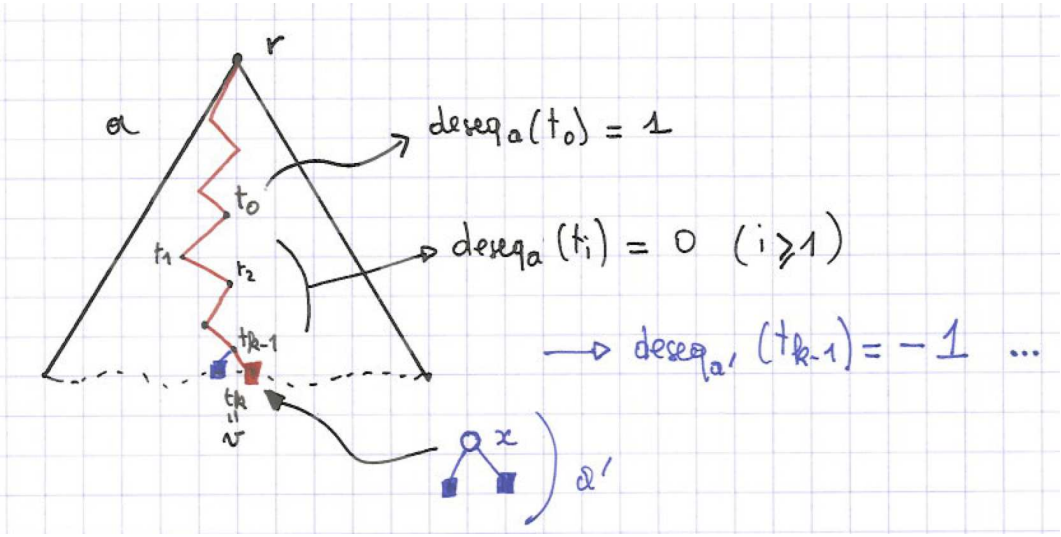

FIGURE 12 – Figure pour la propriété 10

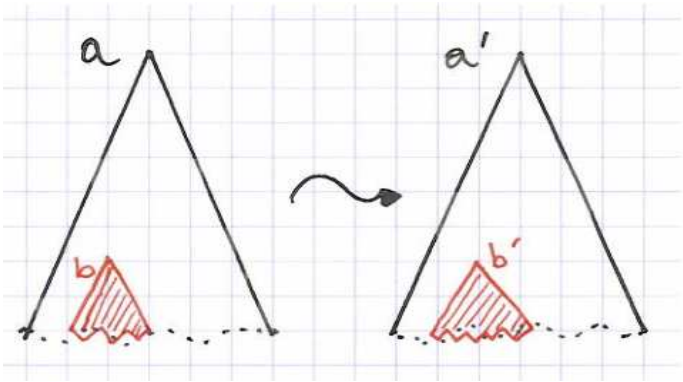

FIGURE 13 – Figure pour le lemme 1

Si b' est un sous-arbre de x, alors supposons que b' est dans $\mathtt{G}_{a'}(x)$. Dans ce cas, la hauteur de $\mathtt{G}_{a'}(x)$ est définie par $\max_s \mathsf{prof}_{\mathtt{G}_{a'}(x)}(s)$: seul compte la longueur maximale menant à une feuille. Or $\mathtt{h}(b) = \mathtt{h}(b')$, cela ne peut pas modifier la hauteur de $\mathtt{G}_{a'}(x)$. □

De plus, nous avons le résultat suivant :

Propriété 11 *Soit a un AVL et x une nouvelle clé tels que l'appel de* `Ajouter-abr`(x, a) *produit un ABR a' qui n'est plus H-équilibré. On considère le plus petit chemin $\rho \overset{def}{=} t_0 \to t_1 \to \ldots \to t_k$ reliant le plus profond sommet t_0 ayant un déséquilibre dans $\{-2, 2\}$ au sommet t_k où x a été ajoutée.*

Alors l'application d'une rotation sur le sous-arbre de racine t_0 permet de rééquilibrer a'.

Preuve :

On note b le sous-arbre de racine t_0 dans a, et b' celui de racine t_0 dans a'.

Supposons que $\mathsf{deseq}_{a'}(t_0) = 2$. La clé x a donc été ajoutée dans le sous-arbre gauche de t_0 (*i.e.* t_1 est le fils gauche de t_0).

On distingue plusieurs cas.

1a - x a été ajoutée dans le sous-arbre gauche de t_1. On est dans la situation décrite par la figure 14. On a $\mathsf{h}(U) = \mathsf{h}(V)$ car $\mathsf{deseq}_a(t_1) = 0$ et $\mathsf{h}(W) = \mathsf{h}(V)$ car $\mathsf{deseq}_a(t_0) = 1$. Clairement une rotation droite sur t_0 suffit pour obtenir un AVL.

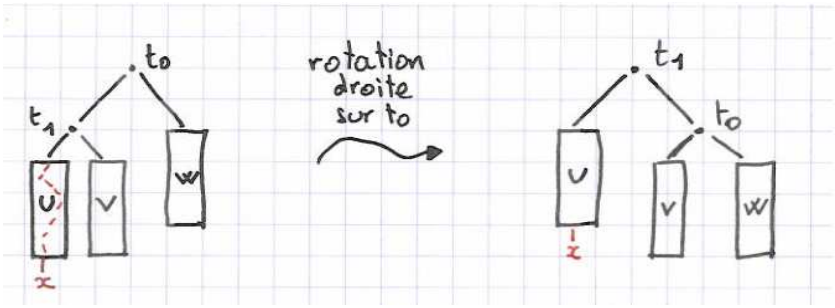

FIGURE 14 – Figure pour le cas 1a

En effet, $\mathsf{h}(b'') = \mathsf{h}(b)$ et donc b'' peut remplacer b dans a sans changer le H-équilibre de a (lemme 1).

NB : la fonction deseq doit être mise à jour. On a clairement $\mathsf{deseq}_{a''}(t_1) = \mathsf{deseq}_{a''}(t_0) = 0$. Ceux entre t_2 et t_k ne changent pas (leur déséquilibre a déjà été changé lors de l'ajout de x , il est dans $\{-1, 1\}$).

1b - x a été ajoutée dans le sous-arbre droit de t_1. On est dans une des deux situations décrites par la figure 15.

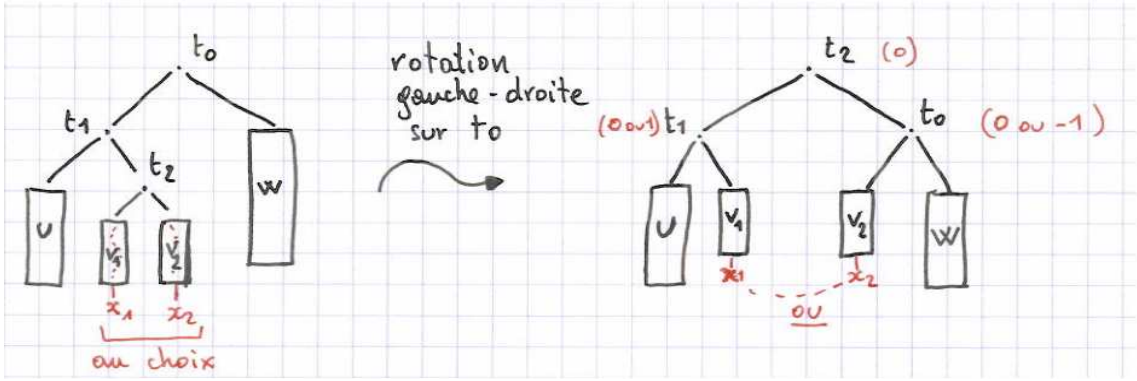

FIGURE 15 – Figure pour cas 1b

On a $\mathsf{h}(U) = \mathsf{h}(W) = \mathsf{h}(V_i) + 1$ (car $\mathsf{deseq}_a(t_1) = 0$ et $\mathsf{deseq}_a(t_0) = 1$). De plus, on a $\mathsf{h}(b) = \mathsf{h}(U) + 2$.

Une rotation gauche-droite sur t_0 suffit pour obtenir un AVL.

En effet, $\mathsf{h}(b'') = \mathsf{h}(U) + 2 = \mathsf{h}(W) + 2 = \mathsf{h}(b)$ et donc b'' peut remplacer b dans a sans changer le H-équilibre de a (lemme 1).

NB : la fonction deseq doit être mise à jour. On a $\mathsf{deseq}_{a''}(t_2) = 0$, $\mathsf{deseq}_{a''}(t_1) \in \{0, 1\}$ et $\mathsf{deseq}_{a''}(t_0) \in \{-1, 0\}$. (idem que pour le cas précédent pour ceux entre t_2 et t_k)

Ensuite il reste à considérer le dernier cas ($\mathsf{deseq}_{a'}(t_0) = -2$) où x est ajouté dans le sous-arbre droit de t_0 avec ses deux sous-cas. C'est symétrique au cas précédent :

2a - x a été ajouté dans le sous-arbre droit de t_1. Voir la figure 16.

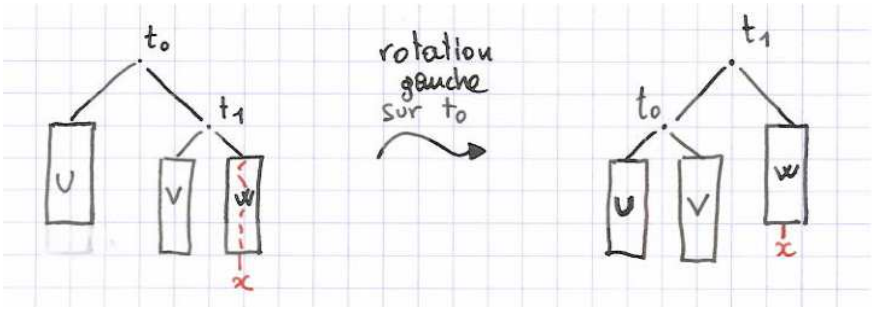

FIGURE 16 – Figure pour cas 2a

On a $\mathsf{h}(U) = \mathsf{h}(V) = \mathsf{h}(W)$. Et $\mathsf{h}(b'') = \mathsf{h}(U) + 2) = \mathsf{h}(b)$.

2b - x a été ajouté dans le sous-arbre gauche de t_1 (voir la figure 17). Et on a $\mathsf{h}(U) = \mathsf{h}(W) = \mathsf{h}(V_i) + 1$. Et $\mathsf{h}(b'') = \mathsf{h}(W) + 2 = \mathsf{h}(b)$.

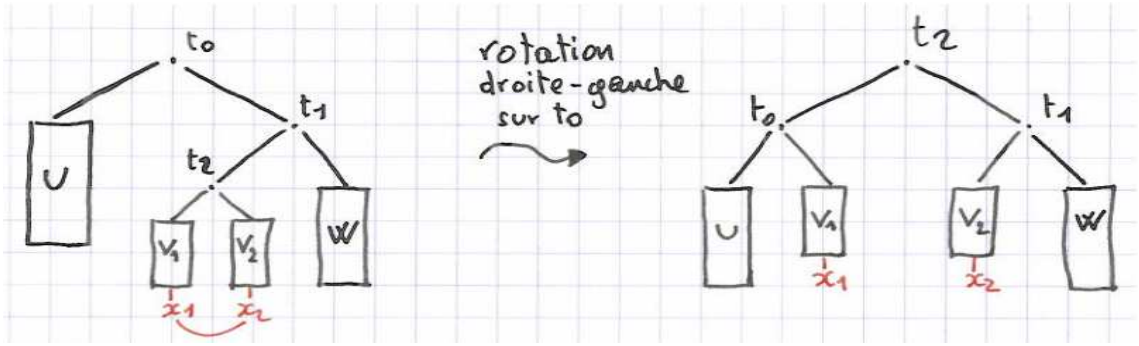

FIGURE 17 – Figure pour cas 2b

□

On en déduit un algorithme d'ajout dans les AVL :

1. on applique l'algorithme d'ajout dans les ABR classiques.
2. Si un problème d'équilibre survient, alors il concerne l'ancêtre t_0 le plus proche de la feuille où x a été ajouté et ayant un déséquilibre de 1 ou -1 dans a.
3. En cas de problème, il suffit d'appliquer localement une rotation en t_0 pour rééquilibrer l'arbre contenant x.

Étant donnée la propriété sur la hauteur des AVL, on en déduit :

Proposition 4 *L'insertion dans un AVL de taille n se réalise en temps $O(\log(n))$. Il suffit au plus d'une rotation (simple ou double) pour rééquilibrer le nouvel arbre.*

5.3 Suppression dans les AVL

La première étape est identique à la procédure utilisée pour la suppression dans les ABR classiques. Ensuite il faut éventuellement rééquilibrer l'arbre. Mais cela peut entraîner plusieurs rotations successives : il va donc être nécessaire de mémoriser tout le chemin entre la racine de a et la place de l'élément supprimé.

Exemple 9 *Un exemple où plusieurs rotations sont nécessaires est le cas des arbres de Fibonacci où la fonction de déséquilibre est toujours égale à* 1 *ou* -1 *pour tous les noeuds.*
A faire en exercice !

Quel est le problème ? Lorsqu'on applique l'algorithme classique de suppression dans un ABR, on remplace la clé à supprimer par la valeur maximale contenue dans le sous-arbre gauche (ou celle minimale du sous-arbre droit) et c'est le noeud qui contient cette clé qui est supprimé : ce noeud est un noeud sans fils droit (sinon sa clé ne serait pas le max) et sa suppression consiste donc à **remonter son sous-arbre gauche** : on remplace donc un sous-arbre (AVL) de hauteur h par un sous-arbre (AVL) de hauteur $h-1$. C'est donc ce problème général qu'il faut étudier.

Remplacement (dans a) d'un sous-avl b de hauteur h par un avl b' de hauteur $h-1$. On distingue plusieurs cas selon le déséquilibre du père x de b et selon la position de b vis-à-vis de x :

1. b est le sous-arbre **droit** de x :
   (a) $\mathsf{deseq}_a(x) = 0$: alors $\mathsf{h}(b) = \mathsf{h}(\mathtt{G}(x))$ et donc $\mathsf{h}(a') = \mathsf{h}(a)$ et $\mathsf{deseq}_{a'}(x) = 1$. La procédure s'arrête : a' est un avl.
   (b) $\mathsf{deseq}_a(x) = 1$: On distingue 3 cas selon le déséquilibre de y, le fils gauche de x :
   (i) $\mathsf{deseq}_a(y) = 0$: On applique une rotation droite (voir la figure 18).

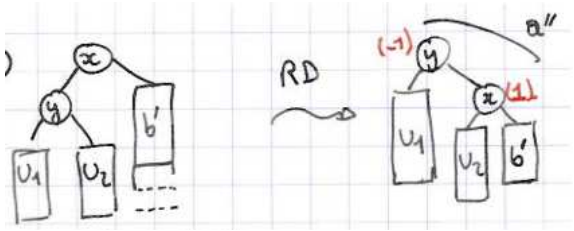

FIGURE 18 – Figure pour cas 1b-i

On a bien $\mathsf{h}(U_1) = \mathsf{h}(U_2) = \mathsf{h}(b') + 1$ et donc $\mathsf{h}(a'') = \mathsf{h}(a') = \mathsf{h}(a)$. C'est terminé.
(ii) $\mathsf{deseq}_a(y) = 1$: On applique une rotation droite (voir la figure 19).

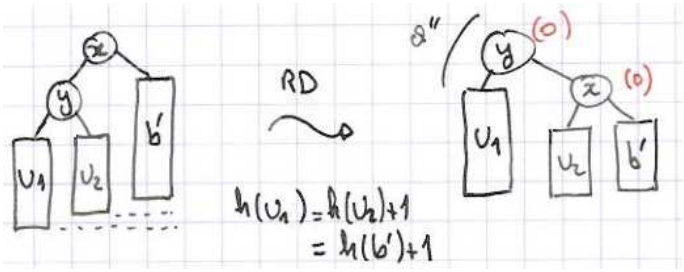

FIGURE 19 – Figure pour cas 1b-ii

Alors $\mathsf{h}(a'') = \mathsf{h}(a) - 1$: on doit donc continuer la procédure de rééquilibrage.
(iii) $\mathsf{deseq}_a(y) = -1$: On applique une rotation gauche-droite (voir la figure 20).
On a $\mathsf{h}(U_1)+1 = \mathsf{h}(U_2) = 1+\max(\mathsf{h}(Z_i)) = \mathsf{h}(b')+1$ et $\mathsf{h}(a'') = \mathsf{h}(U_1)+2 = \mathsf{h}(a)-1$.
On doit continuer la procédure sur le père de x.

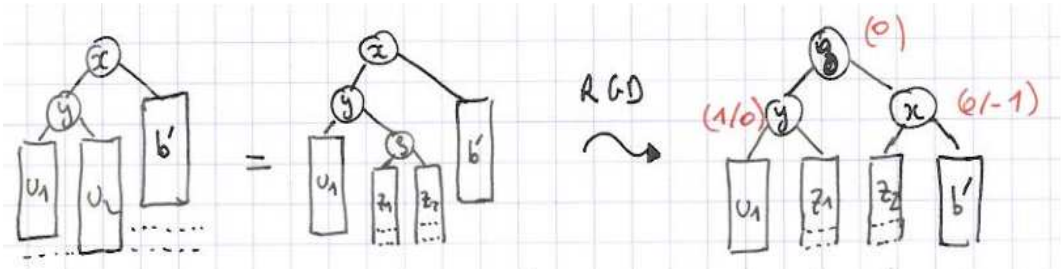

FIGURE 20 – Figure pour cas 1b-iii

(c) $\mathsf{deseq}_a(x) = -1$: On est dans le cas de la figure 21 : aucune rotation n'est nécessaire mais $\mathsf{h}(a') = \mathsf{h}(a) - 1$ et on doit <u>continuer</u> la procédure sur le père de x.

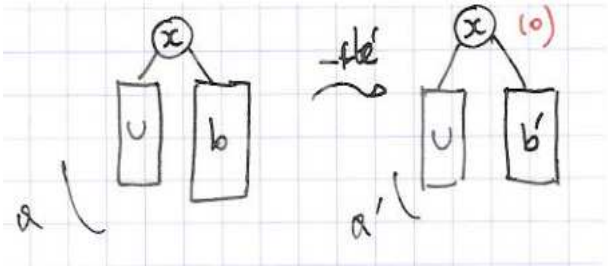

FIGURE 21 – Figure pour cas 1c

2. b est le sous-arbre **gauche** de x :
   (a) $\mathsf{deseq}_a(x) = 0$: On a $\mathsf{h}(a') = \mathsf{h}(a)$ et $\mathsf{deseq}_{a'}(x) = -1$. C'est <u>terminé</u>.
   (b) $\mathsf{deseq}_a(x) = 1$: On est dans le cas de la figure 22.

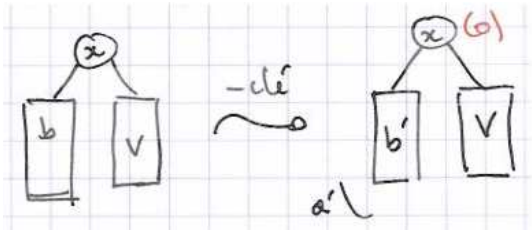

FIGURE 22 – Figure pour cas 2b

   On a donc $\mathsf{h}(a') = \mathsf{h}(a) - 1$ et a' est bien un AVL (aucune rotation n'est nécessaire) mais on doit <u>continuer</u> la procédure sur le père de x.
   (c) $\mathsf{deseq}_a(x) = -1$: On distingue trois cas selon le déséquilibre du fils droit y de x.
      (i) $\mathsf{deseq}_a(y) = 0$: On est dans la situation de la figure 23.
      Après une rotation gauche, on a $\mathsf{h}(a'') = \mathsf{h}(a)$, c'est donc <u>terminé</u>.
      (ii) $\mathsf{deseq}_a(y) = 1$: On est dans la situation de la figure 24.

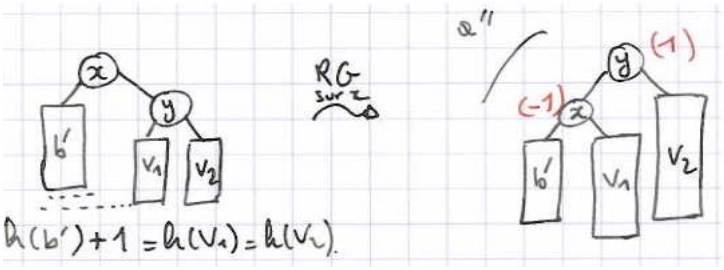

FIGURE 23 – Figure pour cas 2c-i

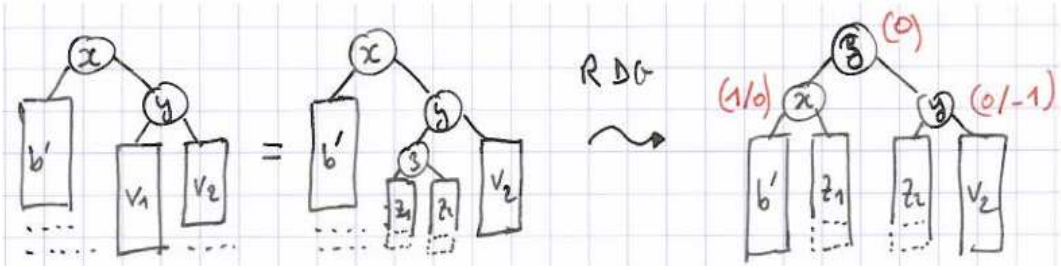

FIGURE 24 – Figure pour cas 2c-ii

Avec $\mathsf{h}(b') = \mathsf{h}(V_1) = 1+\max(\mathsf{h}(Z_i)) = \mathsf{h}(V_2)+1$. Après une rotation droite-gauche, on obtient $\mathsf{h}(a'') = \mathsf{h}(V_2) + 2 = \mathsf{h}(a) - 1$. On doit donc <u>continuer</u>.

(iii) $\mathsf{deseq}_a(y) = -1$: On est dans la situation de la figure 25.

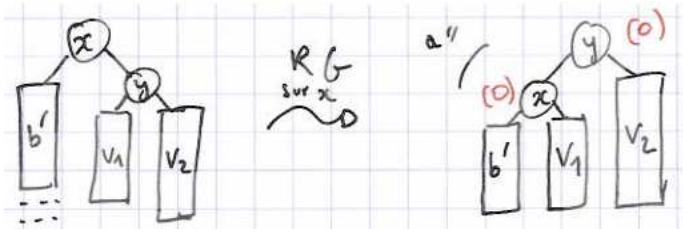

FIGURE 25 – Figure pour cas 2c-iii

On a $\mathsf{h}(V_2) = \mathsf{h}(V_1)+1 = \mathsf{h}(b')+1$. Après une rotation gauche, on a $\mathsf{h}(a'') = \mathsf{h}(a)-1$, on doit donc <u>continuer</u>.

C. Algorithmique dans les graphes

Dans cette partie, beaucoup des algorithmes présentés ainsi que leurs explications viennent de "Introduction à l'algorithmique" de T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Dunod.

1 Définitions

1.1 Graphes orientés

Définition 7 *Un graphe* orienté *G est défini par*
- *un ensemble (fini) S de sommets, et*
- *un ensemble fini de couples (*i.e. *paires ordonnées) de sommets appelés d'arcs ou transitions : $A \subseteq S \times S$.*

La figure 26 contient un exemple de graphe orienté. Les sommets sont q_0, q_1, ..., q_6. Les arcs sont (q_1, q_0), (q_0, q_3), (q_1, q_3), ...

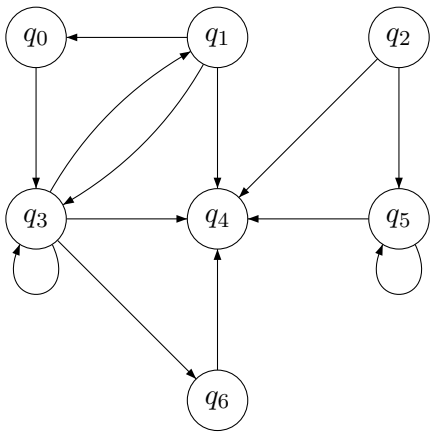

FIGURE 26 – Exemple de graphe orienté.

Étant donné $(u, v) \in A$, on dit que v est adjacent à u et on le note $u \to v$.

Le degré *entrant* d'un sommet u, noté $\mathsf{deg}^+(u)$ est le nombre d'arcs menant à u. Le degré *sortant* de u, noté $\mathsf{deg}^-(u)$ est le nombre d'arcs quittant u.

On définit le degré de u par : $\mathsf{deg}(u) \stackrel{\text{def}}{=} \mathsf{deg}^+(u) + \mathsf{deg}^-(u)$.

Un chemin ρ de v_0 à v_k dans G est une suite de sommets de S $(v_0, v_1, \ldots, v_k)$ tel que $(v_i, v_{i+1}) \in A$ pour tout $i = 0, \ldots, k-1$. Le chemin ρ est de longueur k. On le note $v_0 \to v_1 \to \ldots \to v_k$.

Pour signifier l'existence d'un chemin entre deux sommets u et v, on écrit $u \to^* v$ ou simplement $u \to_G^* v$ lorsque le graphe G est implicite.
Pour signifier l'existence d'un chemin de longueur k entre deux sommets u et v, on écrit $u \to_G^k v$ ou $u \to^k v$.

Un chemin est simple si tous les sommets sont différents.

Un circuit est un chemin de la forme $u \to^p u$ avec $p \geq 1$. Si $p = 1$, on parle de boucle. On dit qu'un circuit est simple lorsque tous les sommets visités – excepté le sommet de départ et d'arrivée – sont différents.

Composantes fortement connexes. Les composantes fortement connexes (CFC) d'un graphe orienté $G = (S, A)$ sont les classes d'équivalence de la relation $\sim$ définie par : $u \sim v \Leftrightarrow (u \to^* v \wedge v \to^* u)$. Ainsi une composante fortement connexe C est un ensemble maximal de sommets tel que si $u, v \in C$ alors $u \to^* v$ et $v \to^* u$: il est toujours possible d'aller de tout sommet u de C à tout autre sommet v de C. Par exemple, pour le graphe de la figure 26, les composantes fortement connexes sont : $\{q_0, q_1, q_3\}$, $\{q_2\}$, $\{q_4\}$, $\{q_5\}$ et $\{q_6\}$.

On dit qu'un graphe orienté G est fortement connexe si il a une unique CFC (l'ensemble S). Dans ce cas, on a $|A| \geq |S| - 1$ (cette borne vient directement du cas non-orienté ; elle est atteinte pour $|S| = 1$ et au-delà on a $|A| \geq |S|$ et cette borne est obtenue pour les cycles).

Un graphe orienté est biparti lorsque S peut être partitionné en deux sous-ensembles S_1 et S_2 tels que $A \subseteq (S_1 \times S_2) \cup (S_2 \times S_1)$: toute transition relie soit un sommet de S_1 à un sommet de S_2, soit un sommet de S_2 à un sommet de S_1.

On dit qu'un graphe orienté G est une arborescence (ou plus simplement un arbre[1]) lorsqu'il est acyclique, que tout sommet a au plus un arc arrivant, et qu'il existe un sommet r tel que pour tout sommet x de G, il existe un chemin de r à x. r est appelée la racine de l'arborescence (et c'est le seul sommet sans arc arrivant, tous les autres en ont exactement un).
Dans le cadre des graphes orientés, une forêt est un ensemble d'arborescences.

1.2 Graphes non-orientés

Définition 8 *Un graphe* non-orienté G *est défini par*
- *un ensemble (fini)* S *de sommets, et*
- *un ensemble fini* A *de paires de sommets* $\{u, v\}$*, appelées arêtes, telles que* $u \neq v$ *(*$A \subseteq \mathcal{P}_2(S)$*).*

La figure 27 contient un exemple de graphe non-orienté. Les sommets sont q_0, q_1, ..., q_6. Les arêtes sont $\{q_0, q_1\}$, $\{q_0, q_3\}$ ...

Étant donnée une arête $\{u, v\} \in A$, on dit que $\{u, v\}$ est incidente à u et v. On appelle degré de u, le nombre d'arêtes incidentes à u.

Remarque : Dans la suite, on écrira souvent l'arête $\{u, v\}$ comme l'arc (u, v) d'un graphe orienté, cela nous permettra d'utiliser des algorithmes valables pour les deux sortes de graphes.

On utilise la même notion de chemin que pour les graphes orientés... Bien sûr, on a la propriété $u \to^* v \Leftrightarrow v \to^* u$ et on écrit aussi $u \leftrightarrow^* v$. Un cycle d'un graphe non-orienté est un chemin $(v_0, v_1, \ldots, v_k)$ tel que $v_0 = v_k$, $k \geq 3$ et $v_1, v_2, \ldots, v_k$ sont distincts. On dit qu'un graphe G sans cycle est acyclique.

Composantes connexes. Un graphe non-orienté $G = (S, A)$ est connexe si $\forall u, v \in S$, on a $u \leftrightarrow^* v$. Les composantes connexes (CC) sont les classes d'équivalence de la relation $\approx$ définie par : $u \approx v \Leftrightarrow u \leftrightarrow^* v$.

On a la propriété suivante :

1. mais cette appellation est plutôt utilisée pour les graphes non orientés comme on le verra plus loin.

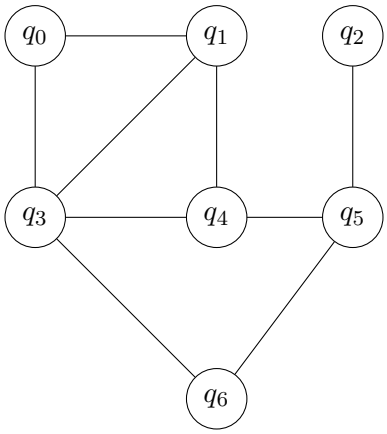

FIGURE 27 – Exemple de graphe non-orienté.

Propriété 12 *Soit $G = (S, A)$ un graphe non-orienté :*

1. *si G est connexe alors $|A| \geq |S| - 1$;*
2. *si G est acyclique alors $|A| \leq |S| - 1$.*

Preuve : (1) la preuve peut se faire par induction sur le nombre de sommets ou sur le nombre d'arêtes. Pour ce premier exemple, nous allons donner les deux preuves :

- sur $|S|$: le case de base $|S| = 1$ est direct ($|A| \geq 0$). Si $|S| = k+1 \geq 2$, alors considérons deux sommets s et s' tels que $\{s, s'\} \in A$ (ils existent sinon G n'est pas connexe). Soit G' le graphe non-orienté où l'on a "fusionné" s et $s' : G' = (S', A')$ avec $S' = S \backslash \{s, s'\} \cup \{s''\}$ et $A' = \big\{\{u, v\} \mid u, v \neq s, s'\big\} \cup \big\{\{s'', u\} \mid \{s, u\} \in A \vee \{s', u\} \in A\big\}$. On a clairement $|A'| \leq |A| - 1$. Et G' est toujours connexe.
  On applique l'hyp. d'induction sur G', on en déduit : $|A'| \geq |S'| - 1$ et donc $|A| - 1 \geq |S'| - 1$, d'où $|A| \geq |S| - 1$.
- sur $|A|$: pour $|A| = 0$, c'est direct car si G est connexe, alors on a $|S| \leq 1$. Pour $|A| = k + 1$. Soit $\{s, s'\}$ une arête de A. Considérons le graphe $G' = (S, A \backslash \{s, s'\})$. Si G' est toujours connexe alors $|A'| \geq |S| - 1$ par h.i. et donc on a le résultat. Sinon, il y a deux composantes connexes (pas plus !) et on peut appliquer l'h.i. sur les deux sous-graphe et obtenir : $|A_1| + |A_2| \geq |S_1| + |S_2| - 2$ et donc $|A| - 1 \geq |S| - 2 \ldots$

(2) la preuve se fait par induction sur $|S|$. Si $|S| = 1$, c'est direct. Supposons $|S| = k \geq 2$. Considérons une arête $\{s, s'\} \in A$ et le sous graphe G' privé de cette arête. Alors le nombre de composantes connexes de G' (noté $\mathsf{NbCC}(G')$) est $\mathsf{NbCC}(G) + 1$: en effet s et s' étaient dans la même CC dans G et ils ne peuvent plus être ensemble dans les CC de G' (car G est acyclique). On a donc au moins deux composantes connexes acycliques dans G' auxquelles on peut appliquer l'h.i. et obtenir $|A_1| + |A_2| \leq |S_1| + |S_2| - 2$ et donc $|A| - 1 \leq |S| - 2$ d'où le résultat. □

Dans le cadre des graphe non orienté, une forêt est un graphe acyclique. Un arbre est une forêt connexe (*i.e.* acyclique et connexe). Les définitions suivantes sont donc équivalentes :

1. G est un arbre,
2. G est connexe et $|A| = |S| - 1$,

3. G est acyclique et $|A| = |S| - 1$,
4. G est connexe à une arête près (en moins),
5. G est acyclique à une arête près (en plus).

Preuve : (1) implique (2) et (3) par une application directe de la propriété précédente.
(2) implique (4) : soit (u, v) une arête de A, et soit $G' = (S, A \backslash (u, v))$. Alors clairement $|A'| < |S| - 1$ et donc G' n'est pas connexe.
(3) implique (5) : soit (u, v) une arête non présente dans A. Soit $G' = (S, A \cup (u, v))$ alors $|A'| > |S| - 1$ et donc G' n'est pas acyclique.
(4) implique (1) : supposons que G a un cycle. Alors on peut enlever une arête et garder la connexité , ce qui contredit l'hypothèse de départ. G est donc acyclique.
(5) implique (1) : supposons que G n'est pas connexe. Alors on peut ajouter une arête entre deux composantes connexes sans créer de cycle. Cela contredit l'hypothèse de départ. Donc G est connexe. □

1.3 Graphes valués

Un graphe valué est un graphe $G = (S, A)$ orienté ou non (on parle soit de graphe orienté valué , soit de graphe non-orienté valué), muni d'une fonction $w : A \to \mathbb{R}$. La fonction w associe un poids, une distance, etc. à chaque arc ou arête de G. La fonction w s'étend naturellement à tout chemin (fini) de G en sommant le poids de chaque arc/arête.

Nous utiliserons ce type de graphe lorsque nous étudierons les algorithmes de plus courts chemins (PCC) ou les algorithmes pour trouver les arbres couvrants minimaux (ACM).

1.4 Représentation des graphes

Liste d'adjacence. Cette représentation est celle la plus couramment utilisée dans les algorithmes (excepté Floyd-Wharshall pour les PCC). Elle convient particulièrement lorsque $|A|$ est nettement plus petit que $|S|^2$.

A chaque sommet u, on associe une liste $L(u)$ de ses sommets adjacents.

Le graphe orienté de la figure 26, se représente par la liste de la figure 28.

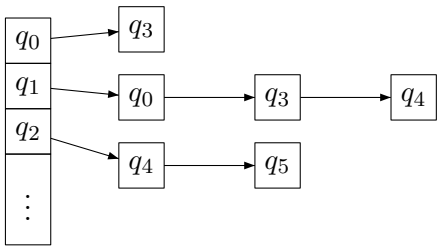

FIGURE 28 – Représentation par liste d'adjacence

La taille de cette représentation est alors en $O(|S| + |A|)$.
On l'étend facilement aux graphes valués en faisant des listes de paires $(d, u) \in \mathbb{R} \times S$.

Matrice d'adjacence. Notons $S = \{u_1, \ldots, u_n\}$. On représente G sous la forme d'une matrice $n \times n$ où le coefficient $\alpha_{i,j}$ vaut 1 si $(u_i, u_j) \in A$ et 0 sinon.

Pour le graphe orienté de l'exemple de la figure 26, on aurait la matrice suivante :

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

La taille de cette représentation est $O(|S|^2)$. Si le graphe est non-orienté, la matrice est symétrique. Notons que ce codage permet aussi de représenter des graphes valués en utilisant la valeur $w(u_i, u_j)$ pour le coefficient $\alpha_{i,j}$ (avec une valeur ∞ lorsque $(u_i, u_j) \notin A$).

2 Parcours de graphes

C'est un problème fondamental dans les graphes. Il est à la base de la question "y a-t-il un chemin entre deux sommets donnés x et y ?"

On va voir plusieurs manière de parcourir un graphe et à chaque fois on gardera la trace de ce parcours sous la forme d'un arbre : pour chaque sommet on notera le sommet à partir duquel il a été découvert.

Parcourir un graphe nécessite de marquer les états visités au fur et à mesure du parcours afin d'éviter de boucler (*i.e.* de redécouvrir un sommet déjà découvert)... Ici nous allons utiliser trois couleurs différentes pour marques les états : le blanc, le gris et le noir.

L'algorithme 8 est une procédure générique de parcours d'un graphe $G = (S, A)$.

Procédure `Parcours`(G)
//$G = (S, A)$
begin
  pour chaque $x \in S$ **faire** Couleur$[x]$:= blanc
  Choisir $s \in S$
  Couleur$[s]$:= gris
  répéter
    Choisir x tq Couleur$[x]$ = gris
    pour chaque $(x, y) \in A$ **faire**
      si *Couleur*$[y]$ = *blanc* **alors** Couleur$[y]$:= gris
    Couleur$[x]$:= noir
  jusqu'à $\forall x.$*Couleur*$[x] \neq$ *gris* ;
end

Algorithme 8 : algorithme de parcours générique

Cet algorithme est non-déterministe : il dépend de la manière de choisir les sommets en début de boucle ainsi que de l'ordre d'énumération des arcs depuis un sommet donné.
Les deux algorithmes classiques de parcours diffèrent précisément par la manière de choisir le sommet x de couleur gris au début de la boucle principale :

- Dans le parcours en largeur, on va choisir le sommet gris *le plus ancien.*
- Dans le parcours en profondeur, on va choisir le sommet gris *le plus récent* (le dernier à avoir été colorié en gris).

3 Parcours en largeur

L'algorithme 9 présente la procédure de parcours en largeur d'un **graphe non-orienté**. (NB : l'algorithme est identique pour les graphes orientés).

```
Procédure PL(G,s)
//G = (S, A)
begin
    pour chaque x ∈ S\{s} faire
        Couleur[x] := blanc ; Π[x] := nil ; Dist[x] := ∞ ;
    Couleur[s] := gris; Π(s) := nil; Dist[s] := 0 ;
    F := File vide                    // File = structure FIFO
    Ajouter(F, s)
    tant que F ≠ ∅ faire
        x := ExtraireTête(F);
        pour chaque (x, y) ∈ A faire
            si Couleur[y] = blanc alors
                Couleur[y] := gris ;
                Dist[y] := Dist[x] + 1 ;
                Π[y] := x ;
                Ajouter(F, y) ;
        Couleur[x] := noir
end
```

Algorithme 9 : algorithme de parcours en largeur

Idée générale. Étant donnés un graphe G et un sommet s, l'algorithme va parcourir les sommets accessibles depuis s en commençant par ceux situés à la distance 1, puis ceux situés à la distance 2, *etc.* De plus, l'algorithme va (1) calculer la distance de chaque sommet à l'origine s, et (2) construire un arbre $G_\Pi = (S_\Pi, A_\Pi)$ contenant les sommets accessibles depuis s et tel que chaque chemin de s à u dans G_Π soit *un plus court chemin de* G.

NB : ici la longueur d'un chemin est le nombre de transitions le long de ce chemin.

On va utiliser les trois couleurs (blanc, gris, noir) pour marquer les états. Le sens de ce coloriage est le suivant :

- blanc : c'est la couleur des sommets non encore découverts (et c'est la couleur de chaque sommet, excepté s, à l'initialisation) ;
- gris : c'est la couleur des sommets déjà découverts et dont les successeurs (immédiats) n'ont pas encore été tous découverts ;
- noir : caractérise les sommets découverts dont tous les successeurs (immédiats) ont aussi été découverts.

NB : deux couleurs seraient suffisantes pour parcourir le graphe (*i.e.* pour assurer sa terminaison), mais on utilise la couleur intermédiaire (gris) pour distinguer les états découverts et dont on a terminé le "traitement" (c.-à-d. l'inspection des successeurs immédiats) de ceux découverts mais dont on continue d'examiner les voisins.

L'arbre G_Π est défini par le prédécesseur (unique) de chaque état dans G_Π : on va utiliser un tableau $\Pi : S \to S$ pour stocker le prédécesseur de chaque sommet découvert. Et on

utilisera nil pour représenter l'absence de prédécesseur. On aura $A_\Pi \stackrel{\text{def}}{=} \{(\Pi(v), v) | \Pi(v) \neq \mathsf{nil}\}$. L'arc $(\Pi(v), v)$ est donc l'arc par lequel v a été découvert. Pour calculer les distances de chaque sommet à l'origine s, on utilise un tableau $\mathsf{Dist} : S \to \mathbb{N} \cup \{\infty\}$.

La figure 29 présente un graphe G et un exemple de résultat obtenu par l'algorithme de parcours en largeur à partir du sommet q_0 (on note les distances calculées à côté des sommets, et les arêtes de G_Π en gras).

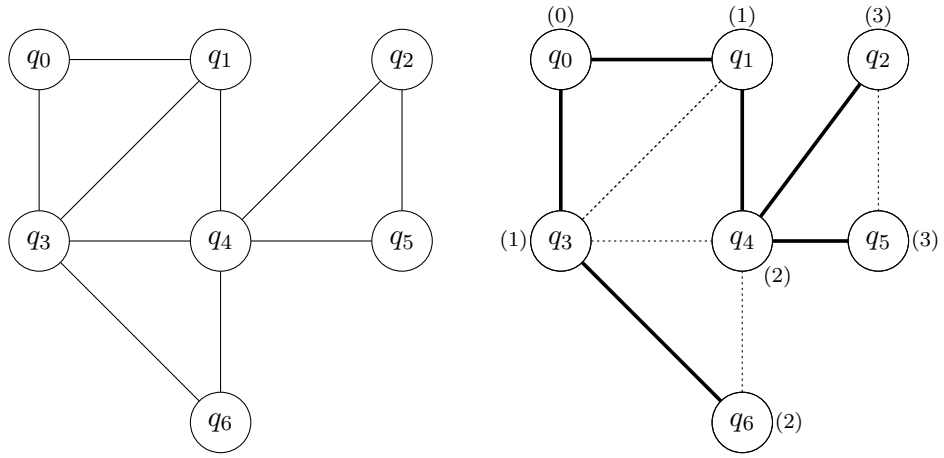

FIGURE 29 – Exemple de parcours en largeur.

Terminaison et complexité de l'algorithme. Tout ajout d'un sommet u dans la file F est conditionné par $\mathsf{Couleur}[u] = \mathsf{blanc}$, or un ajout est toujours suivi par un coloriage en gris et seule la partie "initialisation" colorie en blanc : on en déduit que tout sommet peut être ajouté à (et donc extrait de) la file au plus une fois. Sa liste d'adjacence n'est donc parcourue qu'au plus une fois. On en déduit que l'algorithme termine et qu'en plus la complexité totale de la boucle principale est en $O(|A|)$, auquel on doit ajouter le coût de l'initialisation en $O(|S|)$.
La complexité totale est donc en $O(|S|+|A|)$, *i.e.* en $O(|G|)$: Il s'agit d'un algorithme linéaire dans la taille du graphe.

Plus courte distance. Étant donnés deux sommets u et v, on note $\delta(u, v)$ la longueur d'un plus court chemin de u à v (*i.e.* le nombre minimal d'arêtes pour aller de u à v). Formellement :

$$\delta(u,v) \stackrel{\text{def}}{=} \begin{cases} \min\{p \mid u \to^p v\} & \text{si } u \to^* v \\ \infty & \text{sinon} \end{cases}$$

On a la propriété suivante :

Propriété 13 *Étant donnés $G = (S, A)$ un graphe (non-)orienté et un sommet s, pour tous sommets $u, v \in S$ tels que $(u, v) \in A$, on a : $\delta(s, v) \leq \delta(s, u) + 1$.*

Preuve : Si $\delta(s, u) = k \in \mathbb{N}$, alors il existe un chemin de s à u de longueur k qui peut être prolongé en chemin jusqu'à v avec l'arc (u, v), cela donne un chemin de longueur $k + 1$: tout

plus court chemin de s à v sera donc de longueur au plus $k+1$.
Si $\delta(s,u) = \infty$, alors $\delta(s,v)$ ne pourra pas être strictement supérieur... □

On montre ensuite que les valeurs stockées dans le tableau Dist sont une surapproximation de la distance à s :

Propriété 14 *A tout moment de l'algorithme, on a pour tout sommet u : $\mathsf{Dist}[u] \geq \delta(s,u)$.*

Preuve : La propriété est clairement vraie pour tout sommet u non découvert par la procédure, c'est à dire avec $\mathsf{Dist}[u] = \infty$ durant tout l'algorithme. Montrons que la propriété est vraie pour tout sommet u tel que $\mathsf{Dist}[u] = k$ à la fin de l'algorithme. On le montre par induction sur k.

- $k = 0$: le seul sommet dont la valeur $\mathsf{Dist}[-]$ vaut 0 est s, et sa valeur n'est jamais plus modifiée durant l'algorithme.
- $k+1$: Lorsque $\mathsf{Dist}[u]$ prend pour valeur $k+1$, c'est qu'on a trouvé un sommet v tel que (1) $\mathsf{Dist}[v] = k$, (2) $(v,u) \in A$ et (3) $\mathsf{Couleur}[u] = \mathsf{blanc}$ (et donc $\mathsf{Dist}[u]$ valait jusque là ∞, ce qui garantit la propriété sur le début de l'algorithme). On affecte alors $\mathsf{Dist}[u]$ avec $k+1$ et on change la couleur de u qui ne sera donc jamais plus découvert et dont la valeur $\mathsf{Dist}[u]$ ne bougera plus. Par hypothèse d'induction, on sait que $\mathsf{Dist}[v] \geq \delta(s,v)$ et donc $\mathsf{Dist}[u] \geq \delta(s,v)+1$, or $\delta(s,v)+1 \geq \delta(s,u)$ par la proposition 13.

□

On a la propriété suivante sur le contenu de la file :

Propriété 15 *Lors de l'exécution de PL sur $G = (S,A)$ depuis s, à chaque étape de l'algorithme, si le contenu de la file est de la forme $[v_1, v_2, \ldots, v_k]$ où v_1 désigne l'élément le plus ancien (et donc le premier à être extrait de F) et v_k le plus récent, alors on a :*

- $\mathsf{Dist}[v_i] \leq \mathsf{Dist}[v_{i+1}]$ *pour* $i = 1, \ldots, k-1$
- $\mathsf{Dist}[v_k] \leq \mathsf{Dist}[v_1] + 1$

Preuve : La propriété est vraie au début de l'algorithme et elle est toujours maintenue au cours de la procédure. A la i-ème étape, l'extraction du sommet v_1 peut conduire à l'ajout de ses voisins immédiats u_j (donc à distance $\mathsf{Dist}[u_j] = \mathsf{Dist}[v_1]+1$) dans la file qui devient : $[v_2, \ldots, v_k, u_1, \ldots, u_l]$. Or $\mathsf{Dist}[v_k] \leq \mathsf{Dist}[v_1]+1$ et donc $\mathsf{Dist}[v_k] \leq \mathsf{Dist}[u_i]$. Enfin $\mathsf{Dist}[v_2] \geq \mathsf{Dist}[v_1]$ et donc $\mathsf{Dist}[v_2] \geq \mathsf{Dist}[u_l] - 1$, d'où $\mathsf{Dist}[u_l] \leq \mathsf{Dist}[v_2] + 1$. □

Il y a donc au plus deux types de sommets –du point de vue de la valeur Dist– présents au même instant dans la file F. De plus, une conséquence immédiate de la propriété 15 est que les sommets sont traités par ordre croissant de cette valeur Dist.

On en déduit la correction de l'algorithme de parcours en largeur :

Théorème 1 (Correction du parcours en largeur) *Soient $G = (S,A)$ un graphe non-orienté et $s \in S$ un sommet. L'algorithme $\mathtt{PL}(G,s)$:*

1. *découvre tous les sommets atteignables depuis s et uniquement eux ;*
2. *termine avec $\mathsf{Dist}[v] = \delta(s,v)$ pour tout $v \in S$;*
3. *construit la table Π de telle sorte que pour tout sommet $u \neq s$ atteignable depuis s, il existe un plus court chemin de s à u dans G dont la dernière transition est $(\Pi(u), u)$.*

Preuve : Si un sommet v n'est pas atteignable depuis s, alors $\delta(s,v) = \infty$, et par la propriété 14 on a $\mathsf{Dist}[v] = \infty$ à la fin de l'algorithme. Cela signifie que v n'a jamais été découvert (aucune affectation $\mathsf{Dist}[v]$ et $\Pi[v]$).

Pour les sommets atteignables, on considère l'ensemble $V_k \stackrel{\text{def}}{=} \{v \mid \delta(s,v) = k\}$ pour $k \in \mathbb{N}$. On montre la propriété pour tous les sommets de V_k et pour tout k, par induction sur k. Notons d'abord, comme cela a déjà été souligné, que tout sommet n'est découvert qu'au plus une fois et que ses valeurs pour Dist et Π ne sont modifiées qu'au plus une fois (en plus de l'initialisation).

- $k = 0$: V_0 contient uniquement s. Dès l'initialisation, $\mathsf{Dist}[s] = 0$ et $\mathsf{Couleur}[s] = \mathsf{gris}$, ses valeurs ne seront donc plus modifiées. Il vérifie donc la propriété.
- $k+1$: Soit v un sommet de V_{k+1}. Par la propriété 14, on sait que $\mathsf{Dist}[v] \geq k+1$. Soit x le sommet de V_k t.q. $(x,v) \in A$ qui a été le premier extrait de F. Alors à ce moment, v est encore blanc (sinon v aurait été découvert par un sommet z tel que (1) $z \notin V_k$ *i.e.* $\delta(s,z) \neq k$, et (2) $\mathsf{Dist}[z] \leq k$ car z aurait été extrait de F avant les sommets de V_k, et donc $\delta(s,z) < k$ ce qui contredit $\delta(s,v) = k+1$) et donc on va correctement instancier $\mathsf{Dist}[v]$ avec $k+1$ et le prédécesseur de v par Π, x, sera bien dans V_k.

□

4 Parcours en profondeur

Ici on considère le parcours d'un graphe **orienté**.

Idée de l'algorithme. Le parcours en profondeur consiste à choisir, dans l'algorithme générique 8, le sommet gris découvert le plus récemment. On déroule donc un chemin le plus loin possible. Les sommets gris vont donc former un chemin dans le graphe : le dernier sommet est celui dont on explore les successeurs, puis on reviendra au précédent *etc.*

Les trois couleurs de sommet vont avoir la signification suivante :
- blanc : sommet non encore découvert ;
- gris : sommet découvert mais dont certains descendants n'ont pas encore été découverts ;
- noir : sommet découvert ainsi que tous ses descendants.

Notons qu'ici on parle de descendants (*i.e.* de sommets accessibles par une ou plusieurs transitions) du sommet et non de ses successeurs immédiats. Le sens des couleurs est donc différent de celui utilisé pour le parcours en largeur.

Comme pour le parcours en largeur, on va stocker les arcs ayant permis la découverte de nouveaux sommets dans une table des prédécesseurs G_Π. Pour le parcours en largeur, cette table correspondait à une arborescence, mais pour le parcours en profondeur, G_Π sera plutôt une forêt : un ensemble d'arborescences.

Pour analyser l'algorithme, on va associer une date de coloriage en gris $\mathsf{d}[u]$ et une date de coloriage en noir $\mathsf{f}[u]$ à tout sommet u. Cette date sera un entier entre 1 et $2 \cdot |S|$ (il y a bien $2 \cdot |S|$ opérations de coloriage si l'on omet le coloriage en blanc réalisé à l'initialisation).

Notons que ces dates seront aussi utilisées pour d'autres algorithmes utilisant l'algorithme de parcours en profondeur (composantes fortement connexes, tri topologique).
On dira aussi que $\mathsf{d}[u]$ est la date de *début de traitement de* u, et $\mathsf{f}[u]$ la date de fin de traitement de u.

On a toujours $\mathsf{d}[u] < \mathsf{f}[u]$ et bien sûr :
- avant la date $\mathsf{d}[u]$, u est en blanc ;
- entre $\mathsf{d}[u]$ et $\mathsf{f}[u]$, u est en gris ;
- après la date $\mathsf{f}[u]$, u est en noir.

On considère d'abord l'algorithme 10 d'exploration `PP-Visiter` qui parcourt le graphe –ou plus précisément sa partie non encore visitée– en profondeur depuis un sommet s. Il reste à ajouter la procédure principale : l'algorithme 11.

Notons que contrairement au parcours en largeur où l'on s'intéresse aux sommets atteignables **depuis une origine** s, on souhaite ici **visiter tous les sommets du graphe** G. C'est la raison pour laquelle l'algorithme 11 appelle la procédure `PP-Visiter` sur tous les sommets (non encore découverts).

La figure 30 représente un exemple d'application du parcours en profondeur où l'on appelle, depuis la procédure `PP`, `PP-Visiter`(G, q_0) puis `PP-Visiter`(G, q_2). On indique sur la figure de droite les arcs de la forêt G_Π (notés en gras) et les dates $\mathsf{d}[u]$ et $\mathsf{f}[u]$ pour tout sommet u (par ex. $\mathsf{d}[q_3] = 3$ et $\mathsf{f}[q_3] = 6$).

Parmi les arcs non retenus dans G_Π, on peut distinguer trois catégories :
- les arcs "retour" (marqués "R" sur la figure) sont les arcs (u, v) tels que u est un descendant de v dans G_Π (les boucles en font partie) ;

```
Procédure PP-Visiter(G, s)
//G = (S, A)
begin
    Couleur[s] := gris;
    temps++;
    d[s] := temps;
    pour chaque (s, u) ∈ A faire
        si Couleur[u] = blanc alors
            Π[u] := s ;
            PP-Visiter(G, u) ;
    Couleur[s] := noir ;
    temps++ ;
    f[s] := temps ;
end
```

Algorithme 10 : Procédure de base pour le parcours en profondeur

```
Procédure PP(G)
//G = (S, A)
begin
    pour chaque x ∈ S faire
        Couleur[x] := blanc ;
        Π[x] := nil ;
    temps := 0 ;
    pour chaque x ∈ S faire
        si Couleur[x] = blanc alors
            PP-Visiter(G, x) ;
end
```

Algorithme 11 : Procédure de parcours en profondeur

- les arcs "avant" (marqués "A") sont les arcs (u, v) ne faisant pas partie de G_Π et tels que v est un descendant de u dans G_Π ;
- les arc "transverses" (marqués "T") sont tous les autres...

La forêt G_Π est définie de la manière suivante : $G_\Pi = (S, A_\Pi)$ avec $A_\Pi = \{(\Pi(v), v) \mid v \in S \wedge \Pi(v) \neq \mathsf{nil}\}$. Là encore on écrira $u \rightarrow^*_{G_\Pi} v$ lorsque v est un descendant de u dans G_Π.

Après le premier appel (depuis la procédure PP), de PP-Visiter sur le sommet s, G_Π est une arborescence contenant tous les sommets atteignables depuis s. Après le second appel (depuis PP) sur un sommet s', G_Π est un forêt contenant deux arborescences (la seconde contient des arcs qui permettent d'atteindre tous les sommets accessibles depuis s' dans G et qui ne l'étaient pas depuis s). Après le troisième appel sur s'', G_Π permet d'aller de s'' à n'importe quel sommet accessible depuis s'' dans G qui ne l'était pas depuis s ou s', *etc.*

Complexité. L'initialisation demande un temps en $O(|S|)$. La procédure PP-Visiter est appelée exactement une fois sur chaque sommet (l'opération est conditionnée par "Couleur$[-] =$ blanc" et le sommet concerné est immédiatement colorié en gris ensuite, et seule la phase d'initialisation colorie en blanc). La complexité des boucles "**pour chaque** $(s, u) \in A \ldots$" de tous

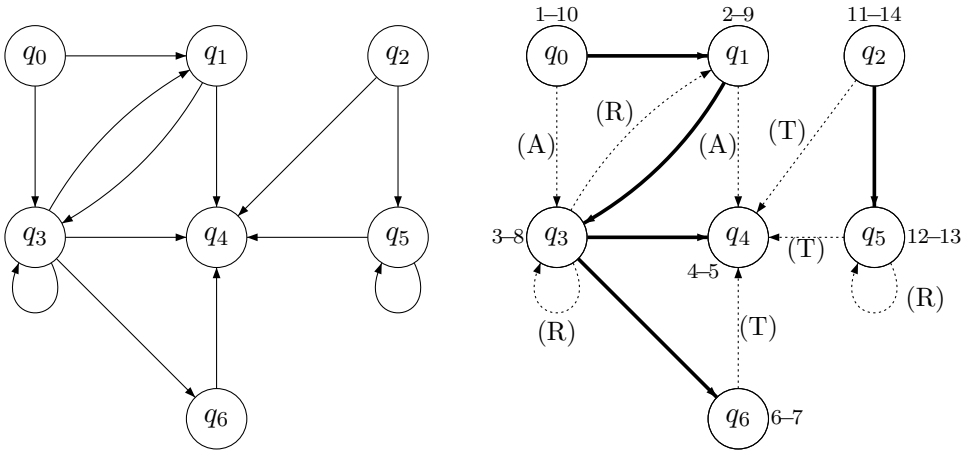

FIGURE 30 – Exemple de parcours en profondeur.

les appels `PP-Visiter` est donc en $O(|A|)$. La complexité totale des appels de `PP-Visiter` est donc en $O(|S| + |A|)$.
On a donc un algorithme linéaire, *i.e.* en $O(|S| + |A|)$.

Correction de l'algorithme. Nous allons maintenant établir une série de propriétés sur les dates $\mathsf{d}[-]$ et $\mathsf{f}[-]$.

Propriété 16 *Pour tous sommets u et v, on a :*
- *soit les intervalles $[\mathsf{d}[u]; \mathsf{f}[u]]$ et $[\mathsf{d}[v]; \mathsf{f}[v]]$ sont disjoints ;*
- *soit $[\mathsf{d}[u]; \mathsf{f}[u]]$ est contenu dans $[\mathsf{d}[v]; \mathsf{f}[v]]$;*
- *soit $[\mathsf{d}[v]; \mathsf{f}[v]]$ est contenu dans $[\mathsf{d}[u]; \mathsf{f}[u]]$.*

Preuve : Supposons $\mathsf{d}[u] < \mathsf{d}[v]$. On découvre donc u avant v. On distingue alors deux cas :
- $\mathsf{d}[v] < \mathsf{f}[u]$: v est découvert alors que u est encore gris, v est donc découvert lors d'un appel de `PP-Visiter` sur un des descendants de u. Étant donnée la procédure `PP-Visiter`, l'exécution de `PP-Visiter`(G, u) n'est alors pas terminée. Après la découverte de v et son coloriage en gris, on va explorer ses successeurs et appeler récursivement la procédure sur eux, puis colorier v en noir et finalement retourner dans les appels en amont... et alors seulement u pourra être colorié en noir. Donc $\mathsf{f}[v] < \mathsf{f}[u]$, on en déduit $\mathsf{d}[u] < \mathsf{d}[v] < \mathsf{f}[v] < \mathsf{f}[u]$.
- $\mathsf{d}[v] > \mathsf{f}[u]$: alors on a $\mathsf{d}[u] < \mathsf{f}[u] < \mathsf{d}[v]$ et donc $\mathsf{d}[u] < \mathsf{f}[u] < \mathsf{d}[v] < \mathsf{f}[v]$ et les deux intervalles sont disjoints.

□

A partir des dates, on peut établir un lien dans la forêt G_Π :

Propriété 17 *Pour tous sommets u et v, on a : $u \rightarrow^*_{G_\Pi} v \Leftrightarrow \mathsf{d}[u] < \mathsf{d}[v] < \mathsf{f}[v] < \mathsf{f}[u]$.*

Preuve : (1) $\Rightarrow$ (2) : soit $u_0 = u \rightarrow_{G_\Pi} u_1 \rightarrow_{G_\Pi} u_2 \ldots \rightarrow_{G_\Pi} u_k = v$ le chemin de u à v dans la forêt G_Π. Pour tout $i = 1, \ldots, k$ on a $u_{i-1} = \Pi(u_i)$, c'est-à-dire que u_i a été découvert lors de l'exécution de `PP-Visiter`(G, u_{i-1}), et sa découverte a conduit à l'appel

récursif de `PP-Visiter`(G, u_i) qui s'est donc terminé avant `PP-Visiter`(G, u_{i-1}). Et donc $\mathsf{d}[u_{i-1}] < \mathsf{d}[u_i] < \mathsf{f}[u_i] < \mathsf{f}[u_{i-1}]$, on en déduit : $\mathsf{d}[u] < \mathsf{d}[v] < \mathsf{f}[v] < \mathsf{f}[u]$.

(2) $\Rightarrow$ (1) : Supposons qu'il existe u et v tels que $\mathsf{d}[u] < \mathsf{d}[v] < \mathsf{f}[v] < \mathsf{f}[u]$ et $u \not\to^*_{G_\Pi} v$. Étant donné u, on choisit le sommet v vérifiant la propriété précédente et **minimisant** $\mathsf{d}[v]$.

Comme u est gris lors de la découverte de v, celui-ci n'a pas été découvert depuis la procédure principale `PP`(G), mais lors de l'exploration des successeurs de u par les appels imbriqués de `PP-Visiter` : donc $\Pi(v) \neq \mathsf{nil}$.

Soit $w = \Pi(v)$. On distingue encore deux cas :

- $\mathsf{d}[u] < \mathsf{d}[w]$: Alors $\mathsf{d}[u] < \mathsf{d}[w] < \mathsf{f}[u]$ (car $\mathsf{d}[w] < \mathsf{d}[v]$ et $\mathsf{d}[v] < \mathsf{f}[u]$) et donc par la propriété 16, on a $\mathsf{d}[u] < \mathsf{d}[w] < \mathsf{f}[w] < \mathsf{f}[u]$ et donc $u \to^*_{G_\Pi} w$ car v a été choisi pour minimiser $\mathsf{d}[-]$. Or comme $w = \Pi(v)$, on a bien un chemin de u à v dans G_Π. Ce qui contredit l'hypothèse de départ.
- $\mathsf{d}[w] < \mathsf{d}[u]$: w est donc découvert avant u. Le traitement de w fait donc des appels à `PP-Visiter`(G, w_i) pour tous les successeurs immédiats de w (dont v fait partie), et **parmi ces appels** il y en a un qui découvre u, **puis** on découvre v (et l'on fait $\Pi(v) := w$). Le traitement de u (*i.e.* l'exécution de `PP-Visiter`(G, u)) est donc terminé lors de la découverte de v. D'où $\mathsf{d}[u] < \mathsf{f}[u] < \mathsf{d}[v] < \mathsf{f}[v]$, ce qui contredit l'hypothèse de départ.

Un tel sommet v n'existe donc pas. □

On peut aussi revenir sur les notions d'arcs "avant", d'arcs "retour" et d'arcs "transverse" en exprimant une propriété sur les dates d et f des sommets concernés :

Propriété 18 *Un arc (v, w) est un...*

1. *arc "avant" ssi $d[v] < d[w] < f[w] < f[v]$;*
2. *arc "retour" ssi $d[w] < d[v] < f[v] < f[w]$; et*
3. *arc "transverse" ssi $d[w] < f[w] < d[v] < f[v]$.*

Preuve : (1) L'arc (v, w) est un arc "avant" ssi w est déjà un descendant de v dans G_Π lorsqu'on examine (v, w) dans `PP-Visiter`. Par la propriété 17, on en déduit $\mathsf{d}[v] < \mathsf{d}[w] < \mathsf{f}[w] < \mathsf{f}[v]$. La réciproque s'obtient de la même manière.

(2) L'arc (v, w) est un arc "retour ssi v est un descendant de w dans G_Π lorsqu'on examine (v, w) dans `PP-Visiter`. Par la propriété 17, on en déduit $\mathsf{d}[w] < \mathsf{d}[v] < \mathsf{f}[v] < \mathsf{f}[w]$. La réciproque s'obtient de la même manière.

(3) Par la propriété 16, on sait que les dates d et f de v et w sont soit de la forme $\mathsf{d}[v] < \mathsf{d}[w] < \mathsf{f}[w] < \mathsf{f}[v]$ ou $\mathsf{d}[w] < \mathsf{d}[v] < \mathsf{f}[v] < \mathsf{f}[w]$ et dans ces deux cas, nous savons par les deux points précédents qu'ils ne sont pas des arcs transverse, soit de la forme $\mathsf{d}[v] < \mathsf{f}[v] < \mathsf{d}[w] < \mathsf{f}[w]$ ou $\mathsf{d}[w] < \mathsf{f}[w] < \mathsf{d}[v] < \mathsf{f}[v]$. Dans ce dernier cas, on peut exclure le premier sous-cas ($\mathsf{d}[v] < \mathsf{f}[v] < \mathsf{d}[w] < \mathsf{f}[w]$) car si à la date $\mathsf{d}[v]$, w est encore banc, alors cela impose que w soit découvert et traité avant que l'arc (v, w) ne soit examiné et donc que cet arc soit un arc avant... Il ne reste donc plus que le dernier cas : $\mathsf{d}[w] < \mathsf{f}[w] < \mathsf{d}[v] < \mathsf{f}[v]$. Inversement si on a cette relation sur les dates, l'arc (v, w) ne peut être ni un arc de G_Π, ni un arc "avant", ni un arc "retour"... □

On en déduit le lien entre les différents sommets gris à chaque itération de l'algorithme :

Propriété 19 *A tout moment de l'algorithme de parcours en profondeur, les sommets gris forment un chemin relié par des arcs de G_Π.*

Preuve : Les sommets gris u_i peuvent être triés par date $\mathsf{d}[-]$ croissantes : $\mathsf{d}[u_1] < \mathsf{d}[u_2] < \ldots < \mathsf{d}[u_k]$. Par la propriété 16, on en déduit :

$$\mathsf{d}[u_1] < \mathsf{d}[u_2] < \ldots < \mathsf{d}[u_k] < \mathsf{f}[u_k] < \ldots \mathsf{f}[u_2] < \mathsf{f}[u_1]$$

Et par la propriété 17, on obtient : $u_1 \rightarrow^*_{G_\Pi} u_2 \rightarrow^*_{G_\Pi} \ldots \rightarrow^*_{G_\Pi} u_k$. Chacune de ces transitions $\rightarrow^*_{G_\Pi}$ est en fait élémentaire (les sommets blancs ne peuvent pas faire partie d'un chemin de G_Π ; et les sommets noirs ne sont pas traversés car ils n'ont pas des sommets gris comme descendants dans G_Π). □

A chaque itération de l'algorithme, on est donc dans une situation du type de celle de la figure 31.

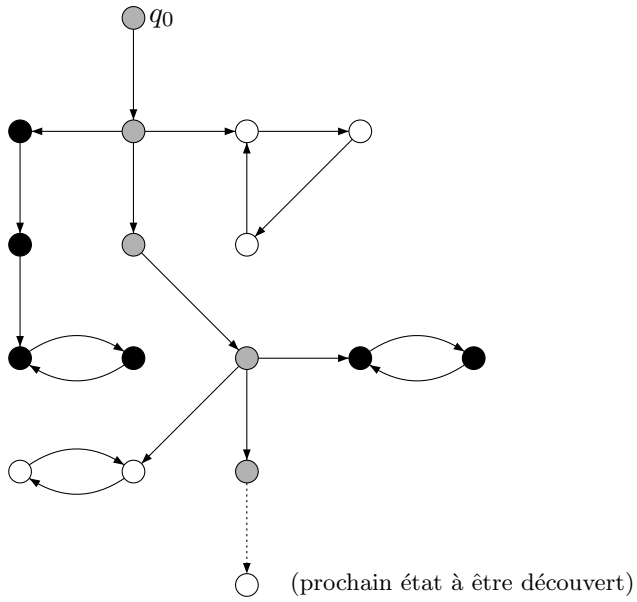

FIGURE 31 – Structure des sommets gris durant l'algorithme de parcours en profondeur.

On en déduit le théorème suivant qui est à la base de la correction de l'algorithme de parcours en profondeur :

Théorème 2 (du chemin blanc) *v est un descendant de u dans G_Π* ***si et seulement si*** *à la date $\mathsf{d}[u]$, le sommet v était atteignable depuis u par un chemin composé uniquement de sommets blancs.*

Preuve : (1) ⇒ (2) : Soit ρ le chemin $u \rightarrow^*_{G_\Pi} v$. Donc tout sommet w de ρ est un descendant de u dans G_Π et d'après la propriété 17, on a $\mathsf{d}[u] < \mathsf{d}[w]$: tout sommet w est donc blanc à la date $\mathsf{d}[u]$.

(2) ⇒ (1) : Supposons que v soit atteignable à la date $\mathsf{d}[u]$ par un chemin blanc ρ depuis u. Supposons que $u \not\rightarrow^*_{G_\Pi} v$ et considérons un tel sommet v **le plus proche possible de** u (sur ρ).

Tout prédécesseur w de v sur ρ est un descendant de u dans G_Π et donc (propriété 17) on $\mathsf{d}[u] < \mathsf{d}[w] < \mathsf{f}[w] < \mathsf{f}[u]$. Considérons le premier w le long de ρ tel que $(w, v) \in A$ (un tel w existe car ρ mène de u à v). Alors v doit être colorié en gris avant la fin du traitement de w

car sinon l'arc (w, v) aurait été considéré lors du traitement de w et v serait un descendant de u dans G_Π. Donc $\mathsf{d}[v] < \mathsf{f}[w]$. D'après la propriété 16, il y a deux cas possibles :

- soit $[\mathsf{d}[v]; \mathsf{f}[v]] \subseteq [\mathsf{d}[w]; \mathsf{f}[w]]$,
- soit $[\mathsf{d}[v]; \mathsf{f}[v]]$ précède $[\mathsf{d}[w]; \mathsf{f}[w]]$.

Dans les deux cas, on obtient que $[\mathsf{d}[v]; \mathsf{f}[v]]$ est contenu dans $[\mathsf{d}[u]; \mathsf{f}[u]]$ et donc, par la propriété 17, on doit avoir $u \rightarrow^*_{G_\Pi} v$ ce qui contredit l'hypothèse de départ. □

Exemple 10 *Nous allons maintenant considérer un autre exemple de parcours en profondeur. La figure 32 décrit un graphe G. Et les figures 33 et 34 donnent deux parcours en profondeur possibles qui se distinguent par l'ordre d'énumération des arcs dans l'algorithme. Cela fournit deux forêts de parcours différentes et change le type des autres arcs ("retour", "avant", "transverse").*

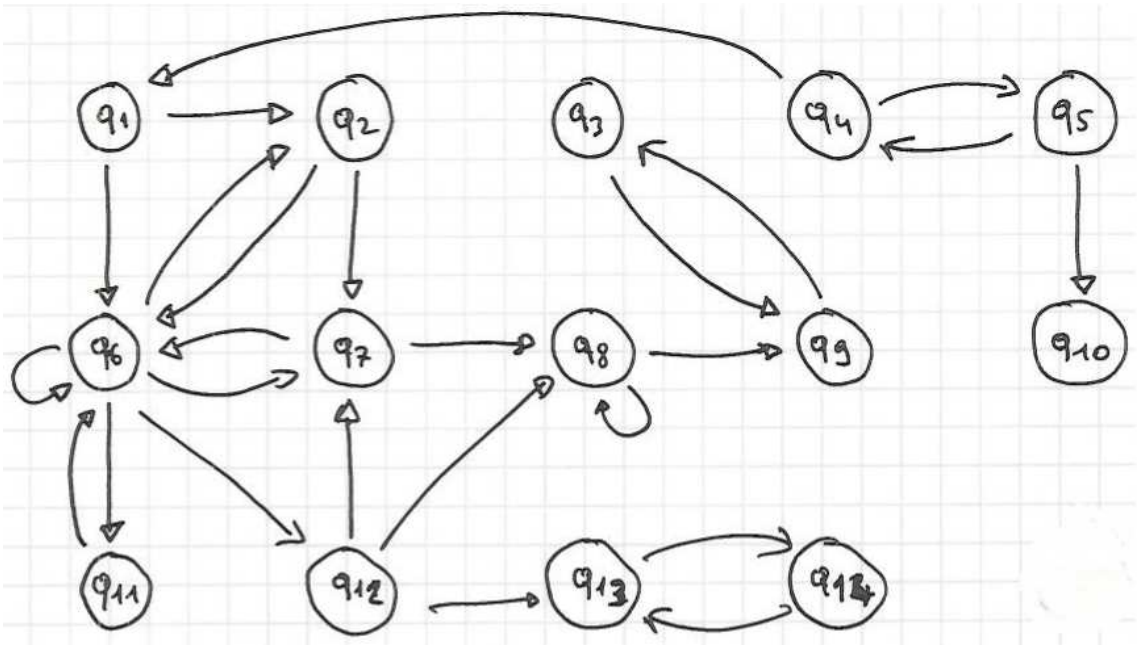

FIGURE 32 – Exemple de graphe pour le parcours en profondeur.

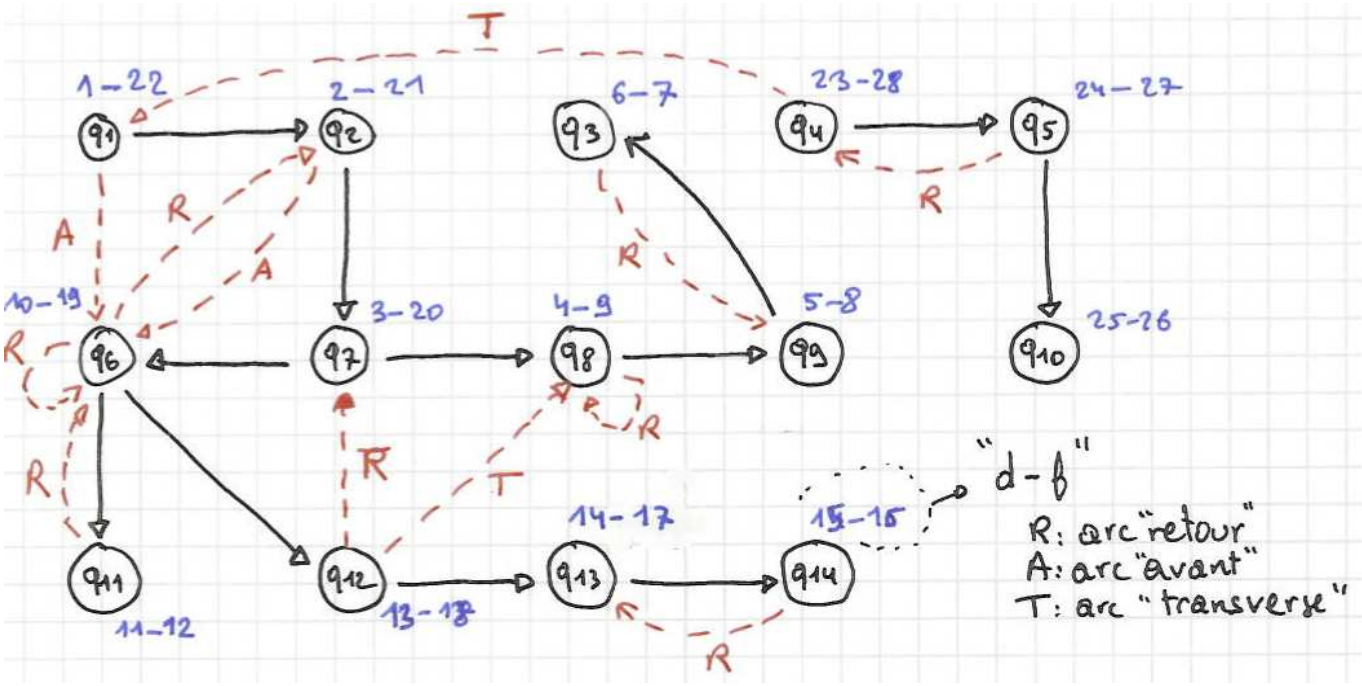

FIGURE 33 – Un parcours en profondeur du graphe de la figure 32

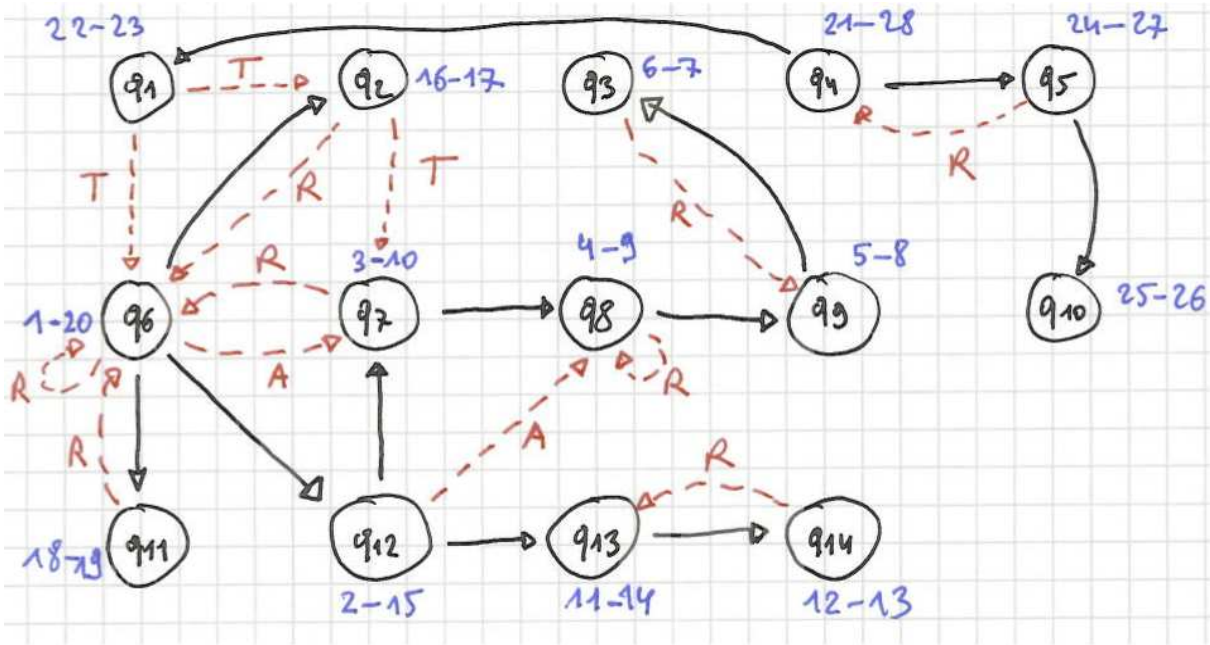

FIGURE 34 – Un parcours en profondeur du graphe de la figure 32

5 Recherche d'une extension linéaire

Ce problème de recherche d'une extension linéaire est souvent appelé tri topologique dans les livres d'algorithmique.

On s'intéresse ici aux **graphes orientés acycliques** (DAG).

Définition 9 *Une extension linéaire d'un graphe orienté acyclique $G = (S, A)$ est un ordre total $\leq$ sur les sommets* compatible *avec A, c'est-à-dire tel que pour tout u et v, on a : $(u, v) \in A \Rightarrow u \leq v$.*

Dans la mesure où G est acyclique, la relation $\rightarrow^*$ engendrée par les arcs de A induit un ordre partiel. Il s'agit donc d'en déduire un ordre total (il y a en général plusieurs solutions possibles).

Applications. On a besoin d'extension linéaire lorsqu'on fait de l'ordonnancement de tâches : on dispose de contraintes de la forme "T_i doit être traitée avant T_j", et on cherche une manière d'exécuter ces différentes tâches dans un ordre "correct". Par exemple :

1. la peinture des murs doit se faire avant le parquet ;
2. l'électricité se fait avant la peinture et la préparation des murs ;
3. la plomberie se fait avant la peinture ;
4. la préparation des murs et du plafond est avant la peinture et après l'électricité ;
5. la pose des meubles se fait après la peinture et après le parquet ;
6. la peinture du plafond se fait avant la peinture des murs ;

On peut représenter ces différentes contraintes de précédence sous la forme du graphe de la figure 35 où chaque arc (u, v) représente la contrainte "u précède v".

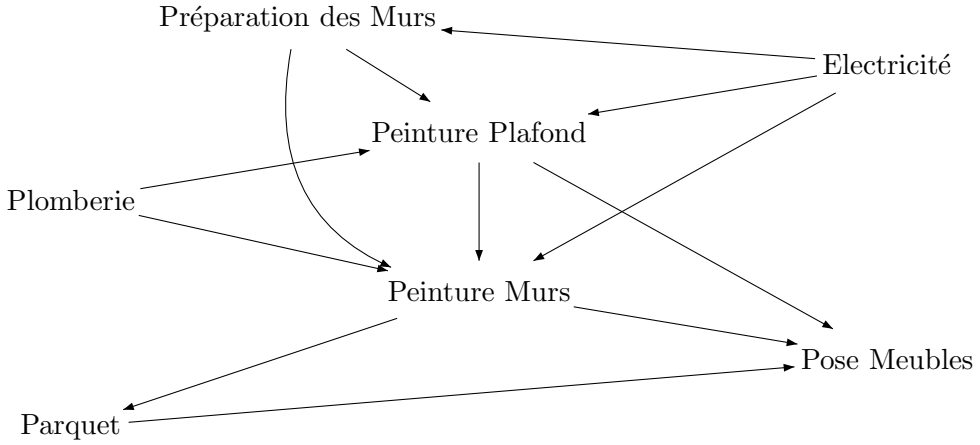

FIGURE 35 – Graphe G_T représentant l'ordre des tâches

Une extension linéaire de G_T fournit une manière d'ordonner les différentes tâches de manière à respecter les contraintes de précédence. Une extension linéaire possible est :

plomberie < électricité < prépa. murs < peinture plafond < peinture murs < parquet < meubles

Un autre possible est :

électricité < prépa. murs < plomberie < peinture plafond < peinture murs < parquet < meubles

Un autre exemple d'application des extensions linéaires concerne l'évaluation d'expressions avec partage de sous-expressions. On peut les représenter sous la forme d'un graphe acyclique (un arbre avec partage) et en déduire un graphe de précédences. Par exemple, l'expression $(a+b)(c+2d)+(a+b)(c-ed)$ peut se représenter sous la forme du graphe de gauche de la figure 36 et on en déduit le graphe de précédence à droite : il faut d'abord évaluer a pour évaluer $a+b$, et $a+b$ avant $(a+b)(c+2d)$, *etc.*

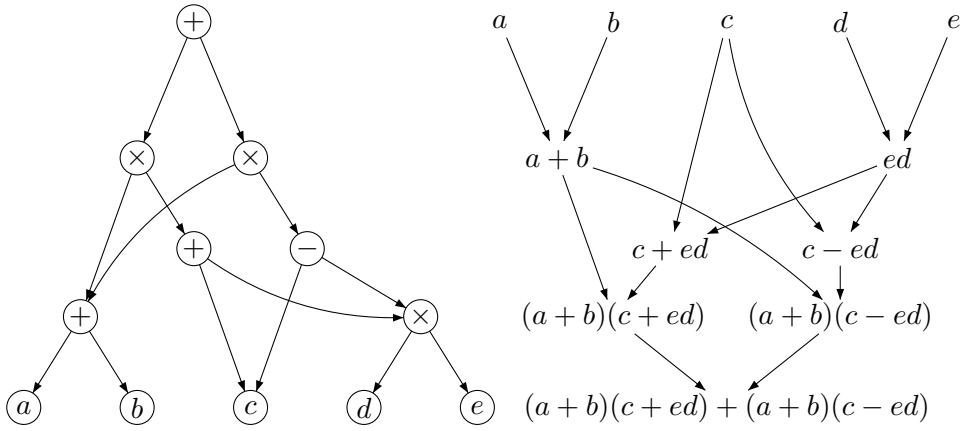

FIGURE 36 – Une expression et son graphe de précédence

Algorithme de recherche d'extension linéaire. L'algorithme suivant, appelé `Recherche-ExLin` dans la suite, permet de trouver une extension linéaire d'un graphe G acyclique :

- Utiliser l'algorithme de parcours en profondeur $\texttt{PP}(G)$
- Empiler dans une pile P chaque sommet lorsqu'il est colorié en noir.

A la fin de l'algorithme, la pile P contient tous les sommets dans l'ordre croissant : le sommet de pile est le sommet minimal pour l'ordre $\leq$ construit.

Une autre manière, équivalente, consiste à appeler la procédure de parcours en profondeur et d'utiliser les dates $\mathsf{f}[-]$ dans l'ordre inverse.

La complexité de l'algorithme `Recherche-ExLin` est donc en $O(|S|+|A|)$.

Une exécution de l'algorithme sur l'exemple précédent pourrait donner les résultats décrits à la figure 37 (les dates $\mathsf{f}[-]$ sont notées en gras).

Avant de considérer la correction de l'algorithme, nous allons d'abord établir une propriété sur les graphes acycliques lors d'un parcours en profondeur :

Propriété 20 *Soit G un graphe orienté. G est acyclique* ***si et seulement si*** *l'algorithme de parcours en profondeur PP ne trouve aucun arc retour*[2]*.*

2. Voir la définition page 41.

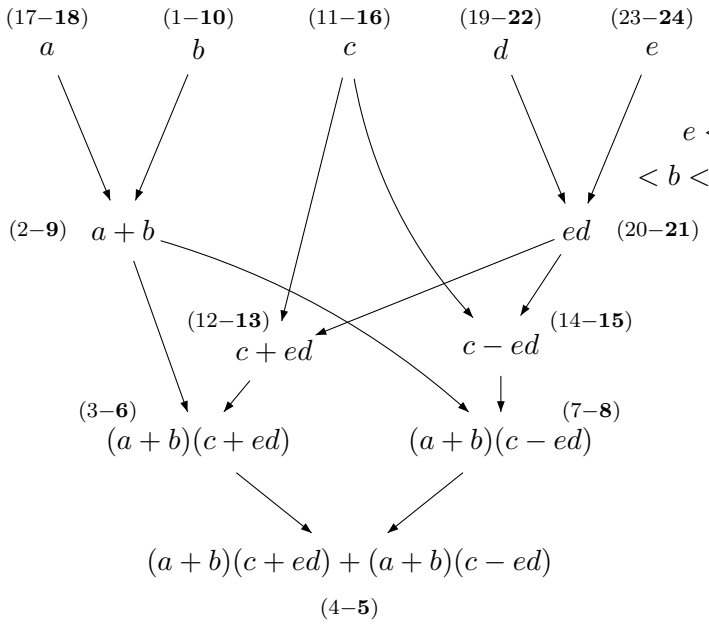

FIGURE 37 – Après le calcul des dates $\mathsf{d}[-]$ et $\mathsf{f}[-]$

Preuve : (1) $\Rightarrow$ (2) : Si (u, v) est un arc retour, alors u est découvert alors que v est toujours en gris. Il y a donc un chemin de sommets gris entre u et v et l'arc (u, v) permet de former un cycle.
(2) $\Rightarrow$ (1) : Supposons qu'il existe un cycle ρ dans G. Soit v le premier sommet de ρ découvert par l'algorithme `PP`. Soit $(u, v) \in A$ l'arc de ρ arrivant en v. Alors à la découverte de v, tous les sommets de ρ sont blancs, et donc par le théorème du chemin blanc, u est un descendant de v dans G_Π et donc (u, v) sera considéré comme un arc retour. □

On va utiliser ce résultat dans la preuve du théorème suivant :

Théorème 3 (Correction de l'algorithme de recherche d'extension linéaire) *Soit G un graphe orienté acyclique. l'algorithme `Recherche-ExLin` donne une extension linéaire de G*

Preuve : Il suffit de montrer que pour tout sommet u et v, on a $(u, v) \in A \Rightarrow \mathsf{f}[u] > \mathsf{f}[v]$.
Soit $(u, v) \in A$. Dans l'algorithme `Recherche-ExLin`, lorsque (u, v) est étudiée, v ne peut pas être gris car alors ce serait un arc retour et G aurait un cycle, donc v est soit blanc (et v sera un descendant de u et donc $\mathsf{f}[v] < \mathsf{f}[u]$) soit noir (et donc $\mathsf{f}[v] < \mathsf{f}[u]$) : dans les deux cas, on vérifie la propriété voulue pour les dates $\mathsf{f}[-]$. □

6 Recherche des composantes fortement connexes ⋆⋆

Comme nous l'avons défini dans la section 1.1, une composante fortement connexe (CFC) $\mathcal{C}$ d'un graphe orienté G est un sous-ensemble *maximal*[3] de sommets de G tel que si u et v appartiennent à $\mathcal{C}$, alors $u \rightarrow_G^* v$ et $v \rightarrow_G^* u$. Cela signifie qu'il existe un chemin pour aller de u à v et un autre pour aller de v à u. Ainsi par exemple, tout cycle dans G contient des sommets appartenant à une même composante fortement connexe.

Ici nous allons décrire l'algorithme de Tarjan (SIAM Journal of Computing, Vol. 1, No. 2, June 1972) pour calculer toutes les composantes fortement connexes d'un graphe. Cet algorithme se base sur l'algorithme de parcours en profondeur et notamment sur le calcul des dates d calculées dans ce parcours.

Exemple 11 *Dans toute cette partie, nous allons suivre l'exemple du graphe G de la figure 32 et de son parcours en profondeur décrit à la figure 33. Ce graphe est rappelé à la figure 38 où les CFC $\mathcal{C}_1, \ldots, \mathcal{C}_7$ sont indiquées, et la forêt de parcours avec les dates $\mathsf{d}[-]$ et $\mathsf{f}[-]$ correspondantes est décrite à la figure 39.*

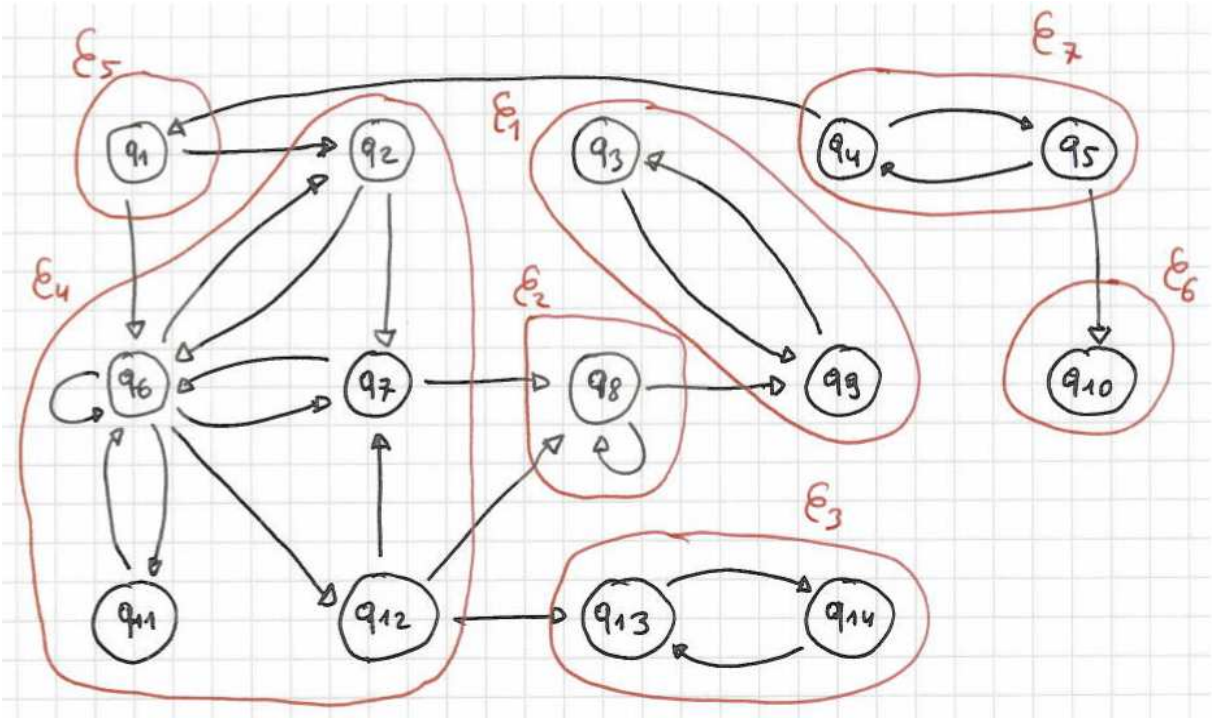

FIGURE 38 – Le graphe exemple pour la section 6.

Nous allons considérer un graphe orienté $G = (S, A)$ et une forêt G_Π obtenue par l'algorithme de parcours en profondeur. Dans la suite, on notera $u \rightarrow_G^* v$ pour dire qu'il existe un chemin entre u et v dans G, et $u \rightarrow_{G_\Pi}^* v$ pour dire qu'il existe un chemin de u à v dans la forêt G_Π. Nous utiliserons aussi la notation $u \leftrightarrow_G^* v$ pour signifier qu'il existe un chemin de u

3. au sens où il n'est pas possible d'étendre ce sous-ensemble sans perdre la propriété $u \rightarrow_G^* v$ et $v \rightarrow_G^* u$ $\forall u, v \in \mathcal{C}$.

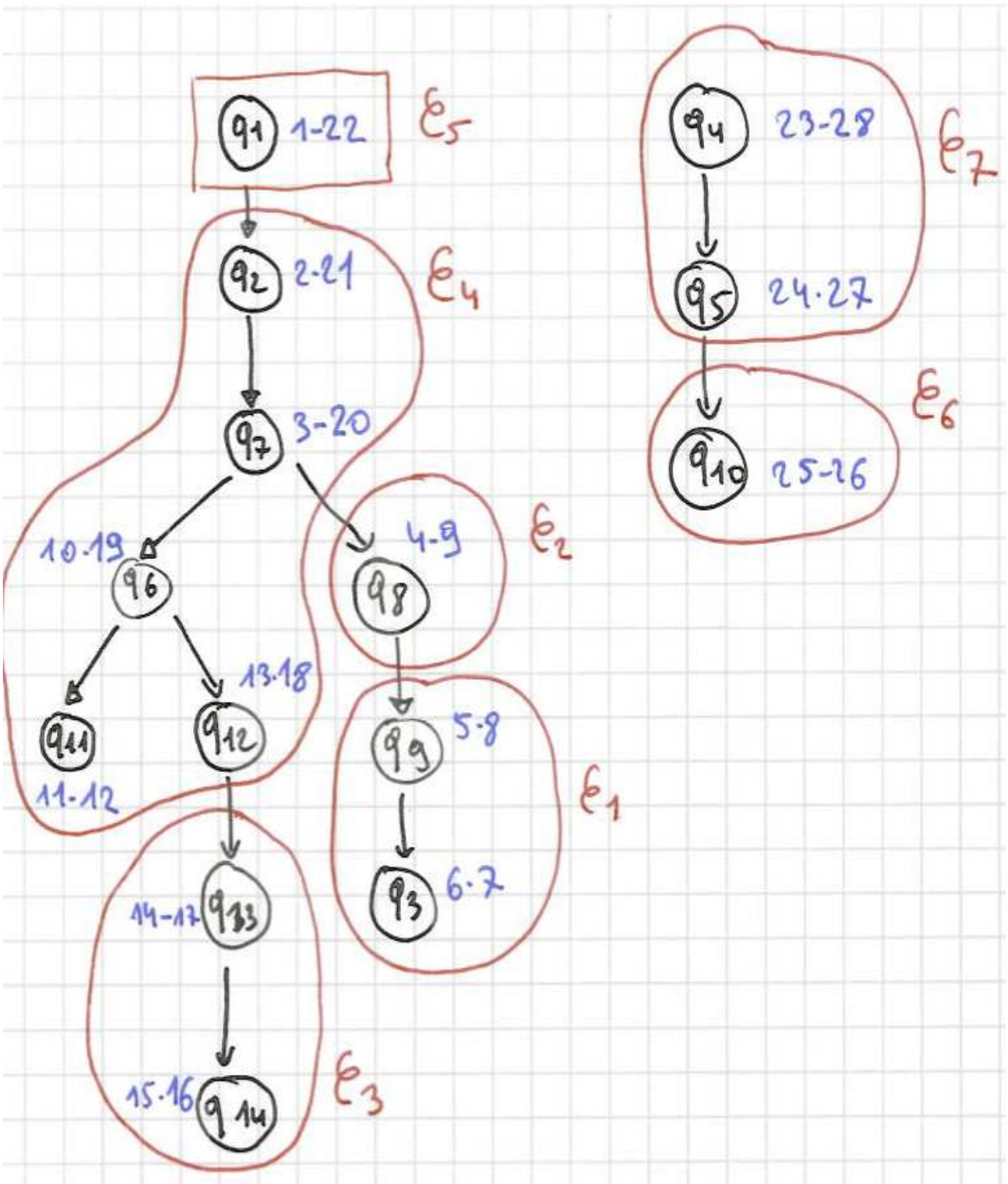

FIGURE 39 – La forêt de parcours de G

à v et un chemin de v à u dans G (c'est-à-dire lorsque l'on a : $u \to_G^* v$ et $v \to_G^* u$). Bien sûr on a : $u \leftrightarrow_G^* v$ ssi u et v sont dans la même CFC de G.

Dans la forêt de parcours G_Π, les sommets d'une même CFC $\mathcal{C}$ n'apparaissent pas n'importe comment : d'une part, ils ont des (au moins un) ancêtres communs dans G_Π et d'autre part, parmi ces ancêtres, celui qui une date $\mathsf{d}[-]$ maximale appartient même à $\mathcal{C}$. Nous avons ainsi la propriété suivante :

Propriété 21 *Soient $v, w \in S$ tels que $v \leftrightarrow_G^* w$,*

1. *alors v et w ont un ancêtre commun dans G_Π ; et*

2. *soit u le sommet tel que (1) $u \to^*_{G_\Pi} v$, (2) $u \to^*_{G_\Pi} w$ et (3) $\mathsf{d}[u]$ est maximal, alors on a : $v \leftrightarrow^*_G u$ et $u \leftrightarrow^*_G w$.*

Preuve :

(1) Soit $\mathcal{C}$ la composante fortement connexe de v et w. Soit $u_\mathcal{C}$ le premier sommet de $\mathcal{C}$ découvert lors du parcours en profondeur (c'est-à-dire ayant le $\mathsf{d}[-]$ minimal). Alors à la date $\mathsf{d}[u_\mathcal{C}]$, il existe un chemin blanc allant de $u_\mathcal{C}$ à tout sommet de $\mathcal{C}$ (ce chemin n'utilise que des sommets de $\mathcal{C}$). Donc en particulier il y a un chemin blanc de $u_\mathcal{C}$ à v et de $u_\mathcal{C}$ à w. Dans les deux cas, on en déduit que $u_\mathcal{C} \to^*_{G_\Pi} v$ et $u_\mathcal{C} \to^*_{G_\Pi} w$ (grâce au théorème du chemin blanc). On obtient donc le premier point de la propriété.

(2) Considérons le sommet u tel que $u \to^*_{G_\Pi} v$ et $u \to^*_{G_\Pi} w$ et $\mathsf{d}[u]$ est maximal (NB : grâce au premier point prouvé ci-dessus, on sait qu'il existe au moins un ancêtre commun à v et w dans G_Π et donc un tel sommet u existe).

Nous avons bien sûr que $u \to^*_G v$ et $u \to^*_G w$ (cela découle directement des hypothèses $u \to^*_{G_\Pi} v$ et $u \to^*_{G_\Pi} w$). Il faut donc montrer l'autre direction.

Comme $\mathsf{d}[u]$ est maximale, on sait aussi que $\mathsf{d}[u] \geq \mathsf{d}[u_\mathcal{C}]$. Toujours avec le théorème du chemin blanc, on obtient $u_\mathcal{C} \to^*_{G_\Pi} u$ et donc :

$$u_\mathcal{C} \to^*_{G_\Pi} u \to^*_{G_\Pi} v \quad \text{et} \quad u_\mathcal{C} \to^*_{G_\Pi} u \to^*_{G_\Pi} w$$

Comme on sait que $u_\mathcal{C}$ est un élément de $\mathcal{C}$, on sait aussi que $v \to^*_G u_\mathcal{C}$ et $w \to^*_G u_\mathcal{C}$. Cela nous donne un chemin de $v \to^*_G u_\mathcal{C} \to^*_G u$ et $w \to^*_G u_\mathcal{C} \to^*_G u$. On a donc bien la propriété recherchée. □

La propriété précédente permet de préciser un peu plus la structure de la forêt de parcours G_Π. En fait, dans G_Π, les sommets d'une CFC $\mathcal{C}$ forment un sous-arbre : il sont bien reliés les uns aux autres. On va appeler la racine de cet arbre, la *racine de* $\mathcal{C}$ et on la notera $u_\mathcal{C}$ dans la suite. On a donc une forêt du type décrit à la figure 40. En effet, on a le corollaire suivant :

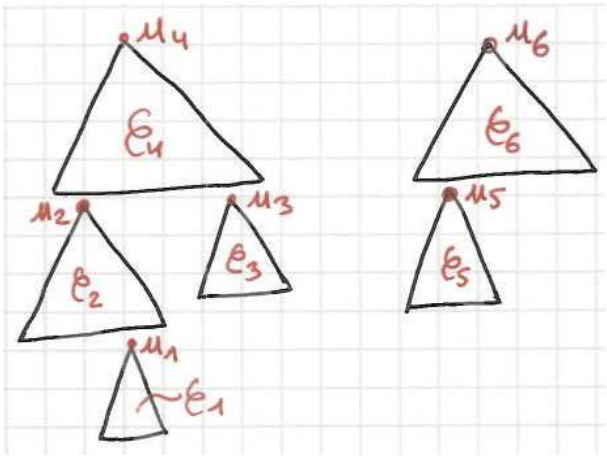

FIGURE 40 – Place des sommets des CFC dans la forêt G_Π.

Corollaire 1 *Soit $\mathcal{C}$ une composante fortement connexe de G. Alors l'arborescence G_Π restreinte aux sommets de $\mathcal{C}$, notée $G_{\Pi|\mathcal{C}}$, est un arbre (couvrant) de $\mathcal{C}$.*

Preuve : Ce corollaire découle directement de la preuve du premier point de la propriété précédente : lorsqu'on découvre le premier sommet $u_{\mathcal{C}}$ de $\mathcal{C}$ tous les autres sommets x de $\mathcal{C}$ sont reliés par des chemins blancs et seront donc des descendants de $u_{\mathcal{C}}$ dans G_Π. De plus, tous les sommets intermédiaires sur le chemin $u_{\mathcal{C}} \to^*_{G_\Pi} x$ sont aussi dans $\mathcal{C}$ car $x \to^*_G u_{\mathcal{C}}$.
$G_{\Pi|\mathcal{C}}$ est donc un arbre (contenant tous les commets de $\mathcal{C}$). □

L'idée de l'algorithme de Tarjan est de calculer un coefficient $\mathsf{r}[x]$ pour chaque sommet $x \in S$ à partir des coefficients $\mathsf{d}[-]$ de manière à ce que pour tout sommet x, on a $\mathsf{r}[x] = \mathsf{d}[x]$ si et seulement si x a été le premier sommet de sa CFC à être découvert dans le parcours en profondeur. De ce fait, les coefficients $\mathsf{r}[-]$ vous nous permettre de trouver les racines des CFC dans la forêt G_Π. Et à partir de ces racines, il sera possible de trouver les autres sommets de ces CFC.

Nous allons maintenant étudier en détail l'algorithme de Tarjan : d'une part, nous allons présenter la définition des coefficients $\mathsf{r}[-]$ et leurs propriétés, et d'autre part l'algorithme pour calculer ces coefficients et les différentes CFC.

6.1 La définition des coefficients $\mathsf{r}(-)$

Dans la suite on suppose disposer des coefficients d et de la forêt G_Π obtenus par un parcours en profondeur sur G.

Pour tout sommet $x \in S$, on définit le coefficient $\mathsf{r}[x]$ de la manière suivante :

$$\mathsf{r}[x] \stackrel{\text{def}}{=} \min\Big(\{\mathsf{d}[x]\} \cup \{\mathsf{d}[w] \mid (1)x \to^*_{G_\Pi} \xrightarrow{T/R} w \wedge (2)\exists u \in S.u \to^*_{G_\Pi} x \wedge u \to^*_{G_\Pi} w \wedge (3)u \leftrightarrow^*_G w\}\Big)$$

Expliquons cette définition... $\mathsf{r}[x]$ est le minimum entre la date de découverte de x et les dates de découverte des sommets w qui :

1. sont accessibles depuis x par un chemin de G_Π prolongé par **un** arc "retour" ou un arc "transverse" (c'est la condition 1), et tels que
2. x et w ont un ancêtre commun dans G_Π, et
3. u et w sont dans la même composante fortement connexe de G.

La figure 41 illustre la situation des sommets w mentionnés dans la définition de $\mathsf{r}[x]$.

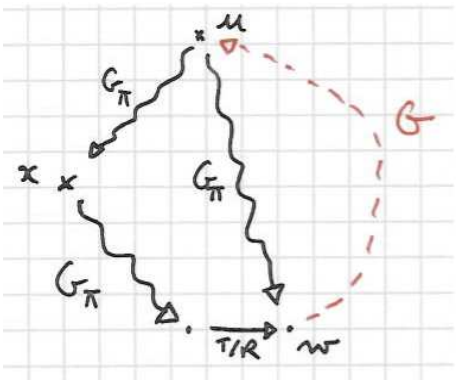

FIGURE 41 – Description des sommets w dans la définition de $\mathsf{r}(x)$.

Une conséquence de cette définition est que le sommets w, le sommet u et le sommet x sont tous dans la même CFC $\mathcal{C}$ du graphe G. On peut reformuler la définition de $\mathsf{r}[x]$ en disant

que c'est le minimum entre $\mathsf{d}[x]$ et les $\mathsf{d}[w]$ tel que w est un sommet de la CFC de x accessible depuis x par un chemin de la forme $x \to^*_{G_\Pi} \xrightarrow{T/R} w$. Les deux définitions sont équivalentes (exercice : pourquoi ?) mais la première formulation correspond plus directement au calcul fait par l'algorithme.

Exemple 12 *La figure 42 présente la forêt de parcours de la figure 39 complétée avec le calcul des coefficients* $\mathsf{r}[-]$ *pour tous les sommets ainsi que les arcs "retour" ou "transverse" utilisés pour les obtenir.*

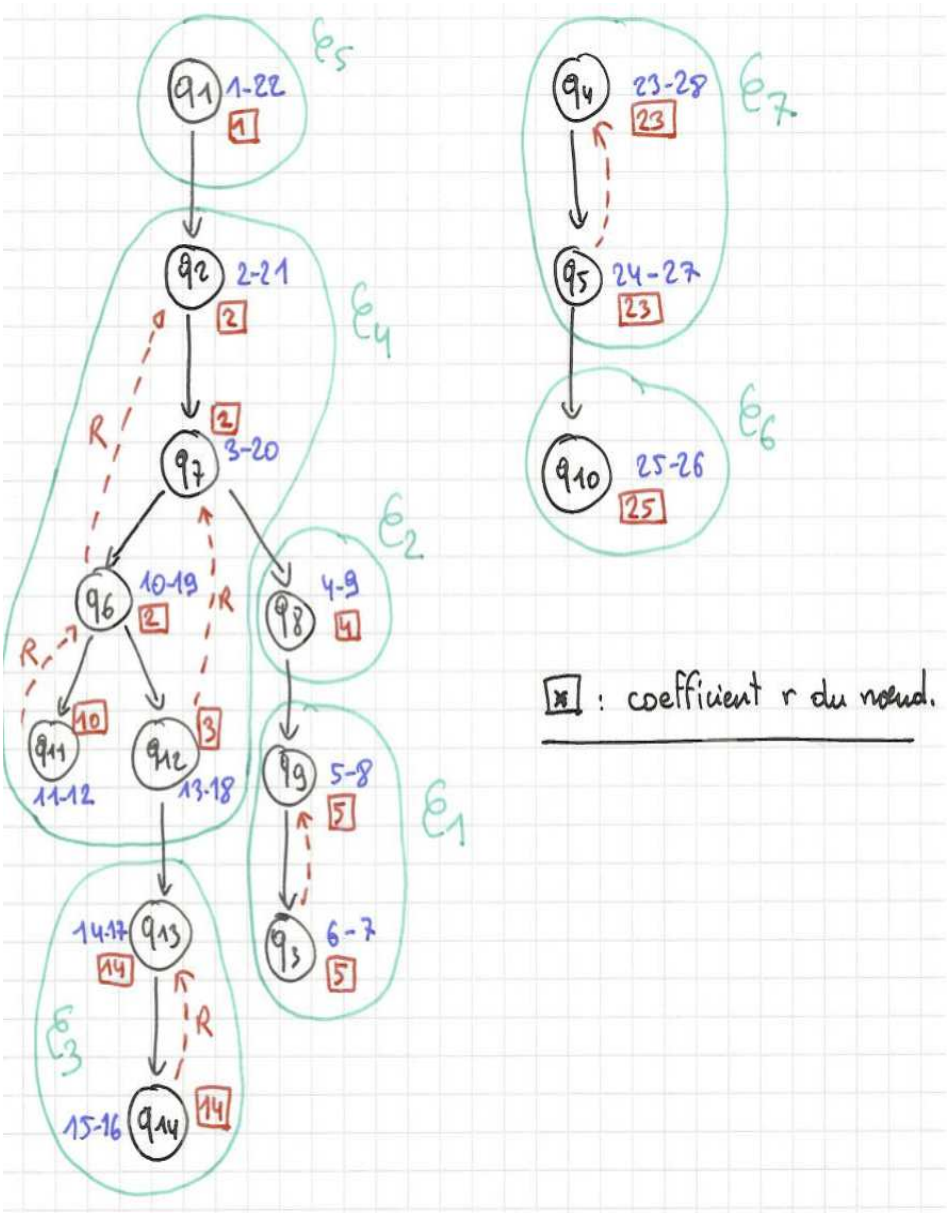

FIGURE 42 – Exemple de calcul des coefficients $\mathsf{r}(x)$.

Puisque w, u et x sont dans la même CFC, on peut déduire la propriété suivante sur la

valeur $\mathsf{r}[x]$:

Propriété 22 *Pour tout sommet x d'une composante fortement connexe $\mathcal{C}$ de racine $u_{\mathcal{C}}$, on a : $d[u_{\mathcal{C}}] \leq r[x] \leq d[x]$*

Preuve : Par définition, on sait que $u_{\mathcal{C}}$ est le premier sommet de $\mathcal{C}$ à être découvert et donc $\mathsf{d}[u_{\mathcal{C}}]$ est inférieure à toutes les $\mathsf{d}[w]$ pour $w \in \mathcal{C}$. Comme la valeur de $\mathsf{r}[x]$ ne dépend que de la valeur $\mathsf{d}[-]$ de certains sommets de $\mathcal{C}$, on a bien le résultat. □

On peut aussi remarquer que si le coefficient $\mathsf{r}[x]$ correspond à un certain $\mathsf{d}[w]$ avec un sommet w accessible depuis x par le chemin $x \rightarrow^*_{G_\Pi} y_1 \rightarrow^*_{G_\Pi} y_2 \cdots \rightarrow^*_{G_\Pi} y_k \xrightarrow{T/R} w$ alors pour tous les sommets intermédiaires y_i, on a : $\mathsf{r}[y_i] = \mathsf{r}[x] = \mathsf{d}[w]$.

On peut à présent énoncer la propriété clé sur la valeur des coefficients $\mathsf{r}[-]$. Lorsque x est la racine de sa CFC, la valeur de $\mathsf{r}[x]$ est $\mathsf{d}[x]$. Mais quand x n'est pas la racine de $\mathcal{C}$, les chemins $\rightarrow^*_{G_\Pi} \xrightarrow{T/R}$ permettent d'atteindre des sommets w ayant un $\mathsf{d}[w]$ inférieur à $\mathsf{d}[x]$ (et donc $\mathsf{r}[x]$ sera bien différent de $\mathsf{d}[x]$) car il existe toujours un chemin de G qui "remonte" à $u_{\mathcal{C}}$ à partir de x (car $u_{\mathcal{C}}$ et x sont dans la même CFC) et on peut trouver un chemin de ce genre qui commence par un préfixe dans G_Π suivi d'un arc "retour" ou "transverse" menant à un sommet de $\mathcal{C}$ ayant une date de découverte inférieure à $\mathsf{d}[x]$. Cette idée est précisée dans la preuve de la propriété ci-dessous :

Propriété 23 *Un sommet x est la racine d'une composante fortement connexe* ***si et seulement si*** *$r(x) = d[x]$.*

Preuve : (1) ⇒ (2) : Conséquence directe de la propriété 22.

(2) ⇒ (1) : Supposons que x n'est pas la racine de $\mathcal{C}$. Montrons que $\mathsf{r}[x] < \mathsf{d}[x]$.

Soit $u_{\mathcal{C}}$ la racine de $\mathcal{C}$. On a donc : $\mathsf{d}[u_{\mathcal{C}}] < \mathsf{d}[x]$ et $u_{\mathcal{C}} \leftrightarrow^*_G x$. A la découverte de $u_{\mathcal{C}}$, tous les sommets de $\mathcal{C}$ sont blancs : il y a donc un chemin blanc reliant $u_{\mathcal{C}}$ à x. On a donc (Th. du chemin blanc) : $u_{\mathcal{C}} \rightarrow^*_{G_\Pi} x$.

Maintenant considérons un chemin ρ de x à $u_{\mathcal{C}}$ dans G. Ce chemin utilise des arcs de G_Π et des arcs "retour", "avant" ou "transverse". On sait que ce chemin ne peut pas être uniquement constitué d'arcs de G_Π car sinon, nous aurions un cycle de x à x dans G_Π (car on a $u_{\mathcal{C}} \rightarrow^*_{G_\Pi} x$) : il y a donc au moins un arc qui n'est pas dans G_Π et nous savons même qu'il y a toujours au moins un arc "retour" ou un arc "transverse" qui permet d'atteindre un sommet z inatteignable avec les seuls arcs de G_Π (NB : tout arc "avant" est facilement remplaçable par une séquence d'arcs de G_Π). Soit (y, z) le premier arc "retour" ou "transverse" de ρ de ce type. On peut donc supposer que ρ a la forme suivante :

$$\rho : \; x \rightarrow^*_{G_\Pi} y \xrightarrow{T/R} z \rightarrow^*_G u_{\mathcal{C}}$$

c'est-à-dire une suite d'arcs de G_Π, suivie d'**un** arc "retour" ou "transverse", puis une suite d'arcs de G. D'après la définition de $\mathsf{r}[x]$, on sait que $\mathsf{r}[x] \leq \mathsf{d}[z]$. Or à la date $\mathsf{d}[x]$, z ne pouvait pas être blanc car sinon il y aurait eu un chemin blanc de x à z et donc on aurait $x \rightarrow^*_{G_\Pi} z$ ce qui est exclu par notre hypothèse sur le choix de (y, z). Donc $\mathsf{d}[z] < \mathsf{d}[x]$. On en déduit $\mathsf{r}[x] < \mathsf{d}[x]$. □

6.2 L'algorithme de calcul des coefficients $\mathsf{r}(-)$

Dans cette section, nous allons présenter l'algorithme de Tarjan et prouver que celui-ci calcule bien les coefficients $\mathsf{r}[-]$ et les composantes fortement connexes. L'algorithme de Tarjan est décrit par les algorithmes 12 et 13. Cet algorithme utilise une structure identique à celle du parcours en profondeur. Dans la suite, nous continuerons de parler de la forêt G_Π même si nous ne la calculons pas explicitement : cette forêt correspond à la manière dont les sommets sont découverts et les arcs énumérés dans les différentes boucles.

Les nouvelles variables utilisées sont :
- *nbcfc* : pour compter le nombre de CFC trouvées ;
- *NumCFC*[−] : un tableau donnant pour chaque sommet le numéro de sa CFC ;
- $\mathsf{r}_A[-]$: un tableau pour stocker les coefficients r ;
- P est une pile pour stocker des sommets de G.

Procédure `Tarjan-CFC`(G)
temps := 0;
nbcfc := 0;
P := `Pile vide`;
pour chaque $x \in S$ **faire**
  Couleur$[x]$:= blanc;
  NumCFC$[x]$:= `undef`;
pour chaque $x \in S$ **faire**
  si *Couleur*$[x]$ = *blanc* **alors** `CFC`(x);
retourner *NumCFC*[]

Algorithme 12 : Algorithme de Tarjan pour un graphe $G = (S, A)$

L'algorithme calcule des coefficients $\mathsf{r}_A[-]$ et nous allons montrer qu'ils correspondent bien aux coefficients $\mathsf{r}[-]$ définis à la section précédente. Le calcul des coefficients $\mathsf{r}_A[-]$ se fait de la manière suivante :

- la pile P va contenir les sommets dont la racine de leur CFC est encore grise (on explore encore ses descendants) : tous ces sommets ont donc des ancêtres communs dans G_Π. Un sommet n'est extrait de P que lorsque sa racine va être colorié en noir et que les instructions 14-18 de la procédure `CFC` sont exécutées.
- l'instruction 8 va permettre de faire "remonter" les valeurs de $\mathsf{r}_A[y]$ à son père x dans G_Π (comme nous l'avons mentionné précédemment, les sommets intermédiaires sur les chemins $x \rightarrow^*_{G_\Pi} \xrightarrow{T/R} w$ utilisés dans la définition de $\mathsf{r}[x]$, ont des $\mathsf{r}[-]$ supérieurs ou égaux à x).
- le test de la ligne 10 caractérise bien les arcs (x, y) de type "retour" ou "transverse" menant à des sommets y ayant des ancêtres communs avec x dans G_Π et appartenant à la même CFC. En effet, on sait que ce n'est pas un arc de G_Π (car y n'est pas blanc) ni un arc "avant" ($\mathsf{d}[y] < \mathsf{d}[x]$). De plus, puisque x et y sont encore dans P, ils ont des ancêtres communs dans G_Π.
  Enfin on peut voir que x et y sont bien dans la même CFC. C'est direct si (x, y) est un arc "retour". Dans le cas d'un arc "transverse", on considère u_x la racine de la CFC de x et u_y celle de y. Ces deux racines sont encore grises (car x et y sont encore dans P) : on a donc soit $u_x \rightarrow^*_{G_\Pi} u_y$, soit $u_y \rightarrow^*_{G_\Pi} u_x$. Dans tous les cas (illustrés à la figure 43), on en déduit qu'il s'agit de la même CFC.

```
1  Procédure CFC(x)
2  temps++; d[x] := temps; Couleur[x] := gris;
3  r_A[x] := d[x];
4  P.Empiler(x);
5  pour chaque (x, y) ∈ A faire
6      si Couleur[y] = blanc alors
7          CFC(y);
8          r_A[x] := min(r_A[x], r_A[y]);
9      sinon
10         si d[y] < d[x] ∧ y ∈ P alors
11             r_A[x] := min(r_A[x], d[y]);
12 temps++; f[x] := temps; Couleur[x] := noir;
13 si r_A[x] = d[x] alors
14     nbcfc++;
15     tant que P ≠ ∅ ∧ d[P.Tete()] ≥ d[x] faire
16         y := P.Tete();
17         P.Depiler();
18         NumCFC[y] := nbcfc;
```

Algorithme 13 : Procédure CFC avec x un sommet de $G = (S, A)$

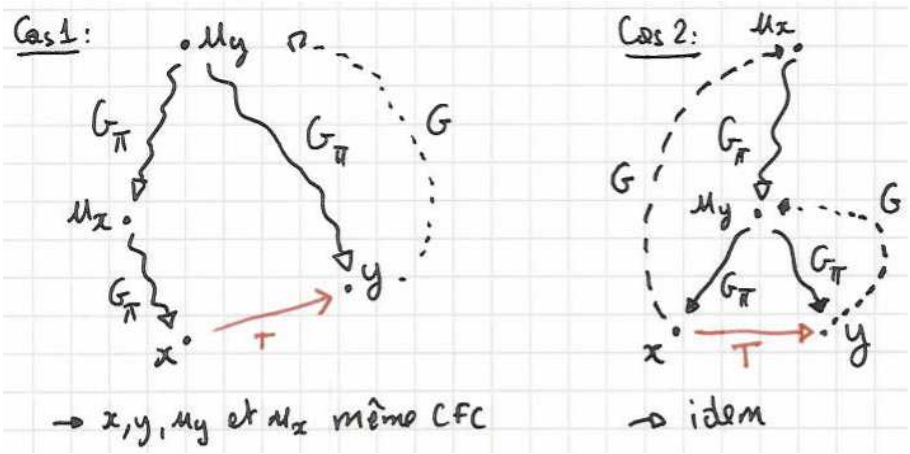

FIGURE 43 – Pourquoi x et y sont dans la même CFC.

Un point clé de la correction va consister à montrer que chaque coefficient $r_A[x]$ est bien égal à $r[x]$, c'est-à-dire que l'algorithme calcule correctement ces coefficients.

Le théorème suivant énonce la correction de l'algorithme :

Théorème 4 *L'algorithme de Tarjan, vérifie les propriétés suivantes :*

1. *à la date $f[x]$, on a : $r_A[x] = r[x]$,*
2. *à la fin du traitement CFC(u_C), tous les sommets y de C sont retirés de la pile P et vérifient : $NumCFC[y] = NumCFC[u_C]$,*

3. *à la fin de* `Tarjan-CFC`(G)*, chaque CFC a reçu un numéro différent : pour deux CFC distinctes* $\mathcal{C}$ *et* $\mathcal{C}'$*, on a* $NumCFC[u_{\mathcal{C}}] \neq NumCFC[u_{\mathcal{C}'}]$.

Preuve : La preuve se fait par induction sur les dates $\mathsf{f}[x]$.

• Cas de base : on considère le sommet x ayant le plus petit $\mathsf{f}[x]$. C'est donc le premier sommet à être colorié en noir par `CFC`. Soit $\mathcal{C}$ la CFC de x. On distingue deux cas :

– x est la racine de $\mathcal{C}$ (qui est donc réduite au seul sommet x) : On va montrer que $\mathsf{r}_A[x] = \mathsf{r}[x]$ et donc $\mathsf{r}_A[x] = \mathsf{d}[x]$. Comme x est le premier sommet à être colorié en noir, c'est une feuille de la forêt G_Π : pour tous les arcs $(x, y) \in A$, on avait $\mathsf{Couleur}[y] \neq \mathsf{blanc}$ à la date $\mathsf{d}[x]$.
Pour s'assurer que $\mathsf{r}_A[x] = \mathsf{d}[x]$, il faut vérifier que l'instruction 11 n'est jamais exécutée, c'est à dire qu'il n'existe pas $(x, y) \in A$ tel que $\mathsf{d}[y] < \mathsf{d}[x]$ et $y \in P$ à la date $\mathsf{d}[x]$. Notons d'abord que y est forcément gris car x est le premier à devenir noir donc un tel arc (x, y) serait un arc "retour" ce qui contredit le fait que $\mathcal{C}$ ne contient que x. Donc un tel arc n'existe pas et $\mathsf{r}_A[x]$ n'est jamais modifié après son initialisation avec $\mathsf{d}[x]$. Donc $\mathsf{r}_A[x]$ est correct.
De plus, après son coloriage en noir, x sera bien dépilé de P (car le test de la ligne 13 sera vrai) et c'est le seul sommet à être retiré de P car le suivant aura un $\mathsf{d}[-]$ forcément inférieur à $\mathsf{d}[x]$ (car découvert avant x). De plus, un nouveau numéro sera choisi pour numéroter la CFC $\mathcal{C} = \{x\}$. Les propriétés sont donc vérifiées.

– x n'est pas la racine de $\mathcal{C}$: Soit $u_{\mathcal{C}}$ la racine de $\mathcal{C}$. Il s'agit maintenant de montrer $\mathsf{r}_A[x] = \mathsf{r}[x] < \mathsf{d}[x]$. Là encore, x n'a aucun successeur dans G_Π. Par contre, il existe un chemin $x \rightarrow^*_G u_{\mathcal{C}}$ dont le premier arc (x, y) est forcément un arc retour (y est dans P et il est gris car seul x est noir) et donc $\mathsf{d}[y] < \mathsf{d}[x]$ et l'instruction 11 sera donc exécutée pour cet arc et modifiera la valeur de $\mathsf{r}_A[x]$. Et finalement, à la date $\mathsf{f}[x]$, x sera laissé dans P.

• Cas général. Soit $x \in S$ appartenant à une CFC $\mathcal{C}$. Par hypothèse d'induction, on sait que tous les sommets ayant un $\mathsf{f}[-]$ inférieur à $\mathsf{f}[x]$ ont été correctement traité. Là encore, on distingue deux cas :

– x est la racine de $\mathcal{C}$: Cela signifie que tous les autres sommets de $\mathcal{C}$ ont déjà été traités (ils sont des descendants de x dans G_Π) et coloriés en noir, et ont donc reçu un $\mathsf{r}_A[-]$ correct (et donc supérieur ou égal à $\mathsf{d}[x]$, voir la prop. 22) et sont encore dans P. Cela implique que les mises à jour de $\mathsf{r}_A[x]$ via l'instruction 8 ne changeront pas la valeur initiale de $\mathsf{r}_A[x]$. Il reste à s'assurer que l'instruction 11 ne sera pas exécutée et donc que $\mathsf{r}_A[x]$ restera avec sa valeur initiale ($\mathsf{d}[x]$).
Considérons un arc $(x, y) \in A$ tel que $\mathsf{Couleur}[y] \neq \mathsf{blanc}$ lors de l'examen de l'arc (instruction 6), $\mathsf{d}[y] < \mathsf{d}[x]$ et $y \in P$. Il y a deux cas :
  – $\mathsf{d}[y] < \mathsf{d}[x] < \mathsf{f}[x] < \mathsf{f}[y]$: l'arc (x, y) est alors un arc "retour" ce qui contredit l'hypothèse "x racine de $\mathcal{C}$".
  – $\mathsf{d}[y] < \mathsf{f}[y] < \mathsf{d}[x] < \mathsf{f}[x]$: alors (x, y) est un arc "transverse" et y appartient à une CFC $\mathcal{C}' \neq \mathcal{C}$. Le sommet y a donc été colorié en noir avant x et c'est aussi le cas de certains de ses prédécesseurs dans G_Π, par exemple la racine de $\mathcal{C}'$ (car sinon cela signifierait que $u_{\mathcal{C}'}$ est toujours gris et donc un ancêtre de x et l'arc (x, y) permettrait de faire un cycle : $u_{\mathcal{C}'} \rightarrow^*_{G_\Pi} x \rightarrow y \rightarrow^*_G u_{\mathcal{C}'}$ et donc $\mathcal{C} = \mathcal{C}'$ et x ne serait pas une racine...). Donc $u_{\mathcal{C}'}$ est déjà en noir en $\mathsf{d}[x]$ et donc, par hypothèse d'induction, les sommets de $\mathcal{C}'$ ont été retirés de la pile en $\mathsf{f}[u_{\mathcal{C}'}]$, le test "$y \in P$" de l'instruction 10 ne peut pas être vrai.

On en conclut que $\mathsf{r}_A[x] = \mathsf{r}[x] = \mathsf{d}[x]$. Et après le coloriage en noir de x, on franchira donc le test de la ligne 13 et on dépilera P jusqu'à trouver x. Avant d'arriver à x on trouvera tous les successeurs de x dans G_Π à l'exception de ceux appartenant à d'autres CFC que $\mathcal{C}$ qui par hypothèse d'induction ont été retirés de P. Il y a donc bien en tête de pile, tous les sommets de $\mathcal{C}$ et seulement eux (jusqu'à x). Ils se verront tous attribuer un même numéro de CFC.

– x n'est pas la racine de $\mathcal{C}$: Soit $u_\mathcal{C}$ cette racine. On va montrer $\mathsf{r}_A[x] = \mathsf{r}[x] < \mathsf{d}[x]$. Considérons le sommet w utilisé pour la définition de $\mathsf{r}[x]$ et soit $\rho : x \rightarrow^*_{G_\Pi} z \xrightarrow{T/R} w$ le chemin menant de x à w. Supposons $x \neq z$, alors c'est par l'instruction 8, que $\mathsf{r}_A[x]$ sera mise à jour car tous les sommets entre x et z le long de ρ auront un $\mathsf{r}[-]$ égal à $\mathsf{d}[w]$ or par hypothèse d'induction, on sait que le calcul du $\mathsf{r}_A[-]$ pour ces sommets est correct... Maintenant supposons $x = z$ (il n'y a pas de transition de G_Π dans ρ) alors $\mathsf{r}_A[x]$ sera bien mis à jour avec $\mathsf{d}[w]$ grâce à l'instruction 11 lors de l'examen de l'arc (x, w) car on aura bien : $\mathsf{d}[w] < \mathsf{d}[x]$ car c'est un arc "retour" ou "transverse" et $w \in P$ car par définition de r, w est dans $\mathcal{C}$ et comme $u_\mathcal{C}$ n'a pas encore été traitée, on a bien $w \in P$ par h.i.

Donc $\mathsf{r}_A[x]$ prendra bien en compte $\mathsf{d}[w]$. De plus, le calcul de $\mathsf{r}_A[x]$ ne prendra pas en compte de "mauvais" sommets. En effet, l'instruction 8 ne prend en compte que des descendants y de x dans G_Π et donc on a $\mathsf{f}[y] < \mathsf{f}[x]$ et donc $\mathsf{r}_A[y] = \mathsf{r}[y]$ par h.i. Et l'instruction 11 est conditionnée par le test de la ligne 10 qui impose $\mathsf{d}[y] < \mathsf{d}[x]$ et $y \in P$, il y a deux cas :

  – soit $\mathsf{d}[y] < \mathsf{d}[x] < \mathsf{f}[x] < \mathsf{f}[y]$: (x, y) est un arc "retour" et donc x et y sont dans la même CFC et il faut bien prendre en compte $\mathsf{d}[y]$ dans le calcul de $\mathsf{r}_A[x]$.
  – soit $\mathsf{d}[y] < \mathsf{f}[y] < \mathsf{d}[x] < \mathsf{f}[x]$: (x, y) est un arc "transverse". Mais ici la condition $y \in P$ assure que la racine de la CFC $\mathcal{C}'$ contenant y n'a pas encore été traitée et est donc encore en gris, il y a donc un chemin entre $u_{\mathcal{C}'}$ et x dans G_Π et l'arc (x, y) suivi du chemin de G reliant y à $u_{\mathcal{C}'}$ est un cycle... x et y sont dans la même CFC et il faut bien prendre en compte $\mathsf{d}[y]$ dans le calcul de $\mathsf{r}_A[x]$.

  Le calcul de $\mathsf{r}_A[x]$ est donc correct. Et le test de la ligne 13 ne sera donc pas vrai et x ne sera pas dépilé.

□

Complexité de l'algorithme. L'algorithme de Tarjan est en $O(|S|+|A|)$. Notons que cela repose sur le fait que chaque sommet ne sera ajouté qu'une seule fois et donc dépilé qu'une seule fois : le coût total de l'exécution des instructions 15–18 sera donc en $O(|S|)$ pour tous les appels de la procédure. A noter aussi que les tests "$y \in P$" peuvent être faits en temps constant si on utilise un tableau de booléens pour garder cette information. L'algorithme est donc linéaire dans la taille du graphe G.

7 Arbres couvrants minimaux

Ici on s'intéresse à la recherche d'*arbres couvrants minimaux* pour des graphes **non-orientés, valués et connexes**.

Soit $G = (S, A, w)$ un graphe non-orienté valué ($w : A \to \mathbb{R}$). On suppose que G est connexe (voir section 1.2). On définit les deux notions suivantes :

- un *arbre couvrant* (S, A') de G est un graphe connexe et acyclique.
- un arbre couvrant (S, A') est dit *minimal* lorsque l'ensemble des arêtes A' minimise la somme $w(A') \stackrel{\text{def}}{=} \sum_{(x,y)\in A'} w(x, y)$.

Rechercher un arbre couvrant minimal (ACM) revient donc à chercher un moyen de relier tous les sommets de G avec un coût total (selon w) minimal. Si S désigne un ensemble de villes, A des routes et la fonction w des distances, alors un arbre couvrant minimal de G représente le réseau routier de longueur totale minimale qui permet de connecter toutes les villes ensemble.
La figure 44 présente un exemple de graphe non-orienté, valué et connexe ainsi que qu'un arbre couvrant minimal (dont les arêtes sont en gras).

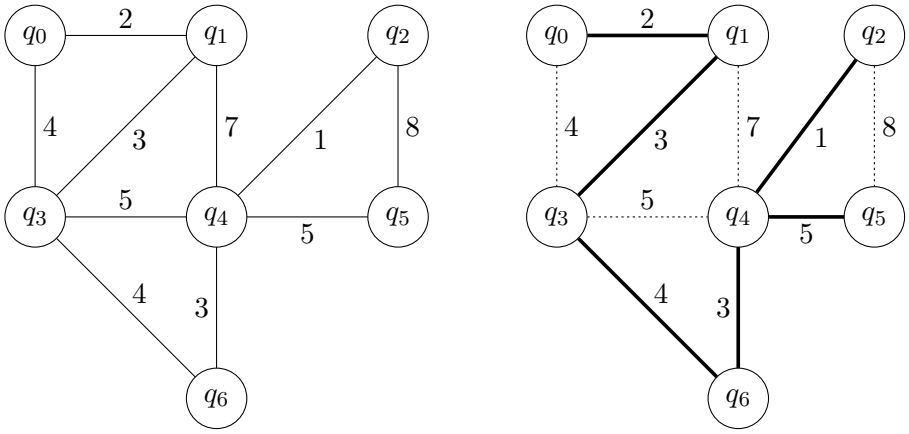

FIGURE 44 – Exemple d'arbre couvrant minimal.

Propriété 24 (Existence d'un ACM) *Tout graphe non-orienté, valué et connexe admet un ou plusieurs ACM.*

Preuve : Tout graphe connexe admet des arbres couvrants (et chacun de ces arbres contient $|S| - 1$ arêtes). Le nombre d'arbres couvrant est fini et il en existe donc au moins un qui minimise $w(A') \stackrel{\text{def}}{=} \Sigma_{(x,y)\in A'} w(x, y)$. □

Un ACM de G se définit par un sous-ensemble d'arêtes de A (de taille $|S| - 1$). Il y a deux algorithmes classiques (et très efficaces) pour faire cela : l'algorithme de Kruskal et l'algorithme de Prim. Ce sont deux algorithmes "**gloutons**" basés sur une idée similaire :

Idée des deux algorithmes : on construit pas à pas un ensemble d'arêtes $A' \subseteq A$ qui est un sous-ensemble d'un ACM pour G. Au début de l'algorithme, A' est vide. Ensuite, à

chaque étape, on doit décider comment choisir une nouvelle arête (u, v) telle que $A' \cup \{(u, v)\}$ est toujours un sous-ensemble d'un ACM pour G (on dit alors que (u, v) est compatible avec A').

Les deux algorithmes se distinguent sur la manière de choisir la nouvelle arête compatible à ajouter à A'.

L'algorithme 14 présente un algorithme générique pour la recherche d'ACM.

Procédure `Recherche-ACM`(G)
`//`$G = (S, A, w)$ `: un graphe non-orienté, valué et connexe.`
begin
  $A' := \emptyset$
  tant que A' *n'est pas un arbre couvrant* **faire**
    Choisir $(u, v) \in A$ t.q. (u, v) est compatible avec A'
    $A' := A' \cup \{(u, v)\}$
  retourner A'
end

Algorithme 14 : algorithme générique de recherche d'un arbre couvrant minimal

Un tel algorithme est correct car : la propriété "A' est un sous-ensemble d'un ACM" est vraie au début de l'algorithme et elle est maintenue à chaque étape. Après la $|S| - 1$-ème étape, A' est un arbre couvrant et c'est un ACM (et donc l'algorithme termine).

Tout le problème réside dans la manière de trouver des arêtes (u, v) compatibles...

Une partition de $G = (S, A, w)$ est une partition (S_1, S_2) de S, c'est-à-dire deux sous-ensembles de S tels que : $S_1 \cup S_2 = S$ et $S_1 \cap S_2 = \emptyset$ (on a bien sûr $S_2 = S \backslash S_1$). Étant donnée une partition (S_1, S_2) de G, on introduit les définitions suivantes :

- une arête (u, v) traverse la partition (S_1, S_2) ssi on a $u \in S_1 \wedge v \in S_2$ ou $u \in S_2 \wedge v \in S_1$ (on dit que (u, v) est une arête traversante de (S_1, S_2)) ;
- (S_1, S_2) respecte $A' \subseteq A$ ssi aucune arête de A' ne traverse (S_1, S_2) ;
- une arête traversante est dite minimale si elle est de poids minimal parmi les arêtes traversantes.

On a le théorème suivant :

Théorème 5 *Étant donnés :*

- $G = (S, A, w)$ *un graphe non-orienté, valué et connexe,*
- $A' \subseteq A$ *tel qu'il existe un ACM de* G *contenant* A'*,*
- (S_1, S_2) *une partition qui respecte* A'*, et*
- (u, v) *une arête traversante minimale de* (S_1, S_2)*,*

alors (u, v) *est compatible avec* A' *(*i.e. *il existe un ACM contenant* $A' \cup \{(u, v)\}$*).*

Preuve : Soit $T \subseteq A$ un ACM contenant $A' : A' \subseteq T$. Si T contient (u, v), on a le résultat. Supposons que $(u, v) \notin T$. Supposons $u \in S_1$ et $v \in S_2$ (NB : (u, v) est traversante).

Comme T est un ACM, il est connexe : il existe donc un chemin ρ allant de u à v. Comme u et v ne sont pas dans le même S_i, le long de ρ, il existe au moins une arête $(x, y) \in T$ traversante pour (S_1, S_2). Supposons $x \in S_1$ et $y \in S_2$. Le chemin ρ peut donc se décomposer de la manière suivante : $u \rightarrow_{\rho_1} x \rightarrow y \rightarrow_{\rho_2} v$.

L'arête (x, y) n'appartient pas à A' car (S_1, S_2) respecte A'. Soit $T' = (T \backslash \{(x, y)\}) \cup \{(u, v)\}$. Clairement $A' \cup \{(u, v)\} \subseteq T'$.

Nous allons montrer que T' est aussi un ACM de G :

– T' est un arbre couvrant : étant donnés deux sommets $z, t \in S$, considérons le chemin π entre z et t dans T. Supposons que (x, y) appartienne à π (sinon π est un chemin de T'). Alors on peut décomposer le chemin de la manière suivante : $z \to_{\pi_1} x \to y \to_{\pi_2} t$. Alors il suffit de remplacer la transition $x \to y$ par le chemin de $x \to_{\rho_1} u$ (dans T) suivi de l'arête (u, v) puis le chemin $v \to_{\rho_2} y$. On obtient donc un chemin (éventuellement non simple) entre z et t dans T', T' est donc connexe. Et c'est bien un arbre car il ne contient que $|S| - 1$ arêtes tout en étant connexe. Donc T' est un arbre couvrant.
– T' a un poids minimal : en effet $w(T') = w(T) - w(x, y) + w(u, v)$. Or $w(u, v) \leq w(x, y)$ car (u, v) est un arc traversant minimal. Et donc $w(T') \leq w(T)$.

(u, v) est donc bien compatible avec A'. □

Les algorithmes de recherche d'ACM sont basés sur le théorème précédent pour choisir les arêtes compatibles. A chaque itération de l'algorithme, l'ensemble A' décrit à une forêt (un ensemble d'arbres) sur S, il induit donc une partition $(C_1, \ldots, C_k)$ où chaque C_i correspond aux sommets d'un arbre de la forêt (*i.e.* une composante connexe de (S, A')). Une arête traversante relie alors des sommets de deux arbres différents... Notons qu'au début de l'algorithme, A' est vide et donc c'est une forêt de feuilles : chaque sommet est isolé des autres et constitue à lui seul un arbre.

Les deux algorithmes se différencient comme ceci :
– Kruskal : à chaque itération A' correspond à un ensemble d'arbres, l'algorithme choisit alors une arête de poids minimal qui relie deux arbres différents.
– Prim : A' correspond à un arbre et un ensemble de sommets isolés. A chaque itération, l'algorithme choisit une arête de poids minimal qui relie l'arbre en cours de construction à un sommet isolé.

7.1 Algorithme de Kruskal

Deux caractéristiques :
– A' décrit une forêt,
– une arête compatible est une arête de poids minimale reliant deux arbres de la forêt.

Pour cet algorithme, on a besoin de tester si deux sommets sont dans le même arbre de A' et de mettre à jour ces informations au fur et à mesure de la construction de A'. Il faut donc disposer d'un type de données manipulant des partitions et permettant les fonctions suivantes :
– tester si deux sommets sont dans le même sous-ensemble. pour cela on va utiliser une fonction Représentant-Ens(s) qui associe à un élément s un représentant canonique de l'ensemble de la partition contenant s.
(ainsi deux sommets x et y seront dans le même arbre ssi Représentant-Ens(x) = Représentant-Ens(y)).
– fusionner deux sous-ensembles. Pour cela on va utiliser une fonction Fusion(s_1, s_2) qui fusionne les deux ensembles contenant s_1 et s_2.

Il faut aussi ajouter un constructeur CréerEnsemble(s) pour créer un singleton contenant s. Ces fonctions correspondent à celles des "Union-Find", une structure de données particulièrement efficaces pour réaliser ces différentes opérations (voir ci-dessous).

L'algorithme 15 décrit l'algorithme de Kruskal en utilisant les fonctions ci-dessus.

L'algorithme est correct car il choisit bien une arête compatible : il suffit d'appliquer le théorème 5 avec la partition $(C_i, S \setminus \bigcup_{j \neq i} C_j)$.

```
Procédure Recherche-ACM-Kruskal(G)
//G = (S, A, w) : un graphe non-orienté, valué et connexe.
begin
    A' := ∅
    pour chaque s ∈ S faire
        CréerEnsemble(s)
    Trier A par poids w(u, v) croissant
    pour chaque (x, y) ∈ A faire
        //On énumère les arêtes dans l'ordre du tri...
        si Représentant-Ens(x) ≠ Représentant-Ens(y) alors
            A' := A' ∪ {(x, y)}
            Fusion(x, y)
    return A'
end
```

Algorithme 15 : algorithme de Kruskal

La complexité de l'algorithme est en $O\Big(|A| \cdot \log(|A|) + F(|S|, |A|)\Big)$ où le premier terme correspond au tri des arêtes et le second correspond au coût de la boucle principale : $F(|S|, |A|)$ désigne le coût de tous les appels aux fonctions sur les partitions (Représentant-Ens et Fusion).

Plus précisément, on sait qu'il y aura $|S|-1$ arêtes ajoutées dans A'. Il y aura donc $|S|-1$ appels à Fusion$(-,-)$ et au pire $2 \cdot |A|$ appels à Représentant-Ens$(-)$.

Si l'on utilise une structure "Union-Find" , on dispose d'opérations particulièrement efficaces : le coût total de m opérations (Représentant-Ens, Fusion et CréerEnsemble) dont n opérations CréerEnsemble a une complexité en $O(m \cdot \alpha(m, n))$ où $\alpha(m, n)$ est une fonction réciproque de la fonction d'Ackermann, $\alpha(m, n)$ croît de façon extrêmement lente et elle est par exemple, majorée par $\log(|A|)$.

On a donc une complexité totale en $O(|A| \cdot \log(|A|))$ car on a $|A| \geq |S| - 1$ (NB : G est connexe).

La figure 45 présente l'ensemble A' après la deuxième et la quatrième étape de l'algorithme de Kruskal exécuté sur le graphe de la figure 44.

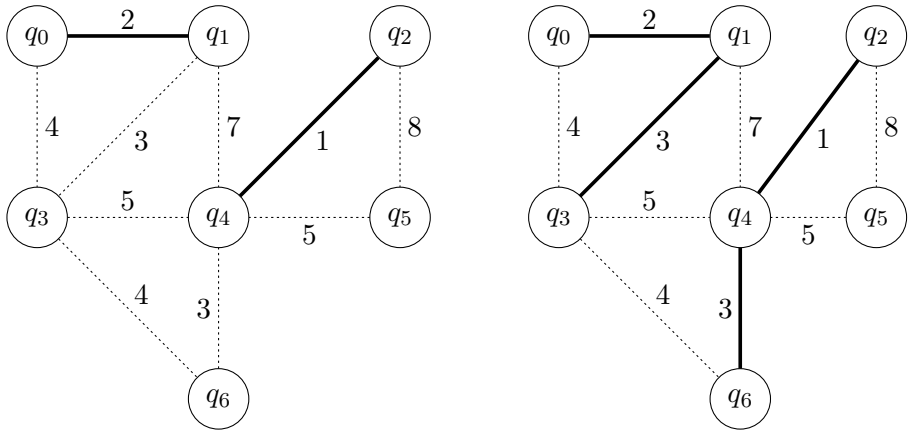

FIGURE 45 – Après les étapes 2 et 4 de l'algorithme de Kruskal.

7.2 Algorithme de Prim

Trois caractéristiques :
- on part d'un sommet donné ;
- à chaque étape de l'algorithme, A' décrit un arbre entouré de sommets isolés ;
- une arête compatible est une arête de poids minimale reliant l'arbre à un sommet isolé.

Pour cet algorithme, on a besoin de trouver le sommet isolé le plus proche de l'arbre en cours de construction. Pour cela, on va utiliser une sorte de file de priorité pour stocker les sommets qui ne sont pas dans encore dans l'arbre. **La priorité k d'un sommet de la file correspondra à sa distance "élémentaire" (c'est-à-dire par une seule arête) par rapport à l'arbre en construction.** Cette file est munie des opérations suivantes :
- Extraire-Min(F) qui extrait de la file F le sommet le plus proche de l'arbre en construction,
- MaJ-F-Prim($F, \mathsf{d}, G, s, \Pi,$ `IndiceDansF`) qui met à jour la file de priorité F, la fonction de priorité d et la fonction de prédécesseur Π (voir l'algorithme 17) après l'ajout de s dans A' : les arêtes (s, u) peuvent rapprocher le sommet u de l'arbre A' et donc modifier $\mathsf{d}[u]$ et la place de u dans la file.

Il faut aussi deux structures complémentaires à la file :
- $\Pi[s]$: une fonction prédécesseur Π qui indiquera par quelle arête un sommet a été choisi comme étant le plus proche de l'arbre.
- `IndiceDansF`$[s]$: est un tableau d'entiers qui donne l'indice d'un sommet dans la file (une file est représentée par un tableau, cf le cours sur les tas) présent dans F (et -1 si le sommet n'est pas dans F). Ce tableau permet notamment de tester efficacement la présence d'un élément dans F.

On utilise aussi un constructeur File(S, d) qui construit la file avec les sommets de S en fonction de la fonction de priorité d.
Remarque : La fonction MaJ-F-Prim n'est pas standard pour les files de priorité mais elle s'implémente assez facilement et efficacement (voir ci dessous).

L'algorithme 16 décrit l'algorithme de Prim en utilisant les fonctions ci-dessus.

L'algorithme est correct car il choisit bien une arête compatible : il suffit d'appliquer le théorème 5 avec la partition $(S \backslash F, F)$ où F désigne ici les sommets de la file F (c'est-à-dire ceux qui ne sont pas reliés à l'arbre en construction).

La figure 46 présente l'ensemble A' après la deuxième et la quatrième étape de l'algorithme de Prim exécuté sur le graphe de la figure 44.

Implémentation de la file. La complexité de l'algorithme est directement liée aux opérations sur la file. Nous donnons ici leur algorithme. L'idée est, comme pour les files de priorité classiques, d'utiliser un arbre binaire parfait pour représenter la structure, et on code cet arbre binaire avec un tableau ($T[2i]$ correspond au fils gauche de $T[i]$ et $T[2i+1]$ est le fils droit...). On complète cet arbre par le tableau d'entiers IndiceDansF$[-]$ décrit précédemment.

Un appel de MaJ-F-Prim(F, d, G, s, Π) prend un temps en $O(n_s \cdot \log(|F|))$ où n_s désigne le nombre d'arêtes contenant le sommet s.

La complexité totale de l'algorithme de Prim se décompose de la manière suivante :
- la construction de la file requiert un temps en $O(|S|)$,
- chaque appel de ExtraireMin se fait en $O(\log(|S|))$, et

Procédure `Recherche-ACM-Prim`(G, s_0)
`//`$G = (S, A, w)$ `: un graphe non-orienté, valué et connexe.`
`//`$s_0 \in S$ `: un sommet ‘‘point de départ’’.`
begin
  pour chaque $s \in S$ **faire**
    $\Pi[s] :=$ nil
    $\mathsf{d}[s] := \begin{cases} 0 & \text{si } s = s_0 \\ \infty & \text{sinon} \end{cases}$
  $A' := \emptyset$
  $F :=$ File$(S, \mathsf{d}, \mathsf{IndiceDansF})$ `//Construit` F `et initialise` IndiceDansF
  tant que $F \neq \emptyset$ **faire**
    $s :=$ Extraire-Min(F)
    IndiceDansF$[s] := -1$
    si $s \neq s_0$ **alors** $A' := A' \cup \{(\Pi(s), s)\}$
    MaJ-F-Prim$(F, \mathsf{d}, G, s, \Pi, \mathsf{IndiceDansF})$
  return A'
end

Algorithme 16 : algorithme de Prim

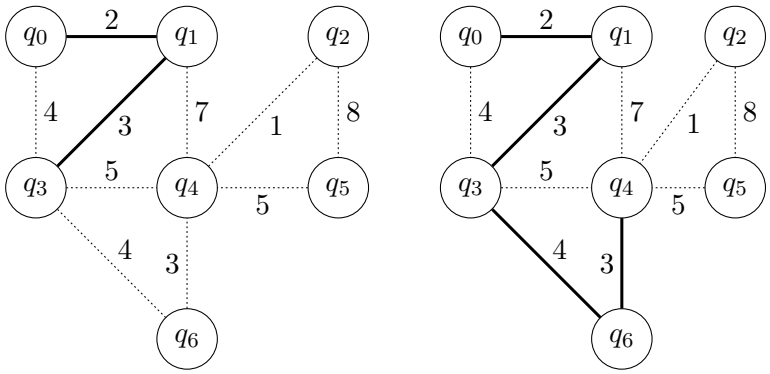

FIGURE 46 – Après les étapes 2 et 4 de l'algorithme de Prim exécuté depuis q_0.

– le coût total des appels de MaJ$(F, \mathsf{d}, G, s, \Pi)$ est en $O(|A| \cdot \log(|S|))$.

On peut majorer le tout avec $O\Big(|S| \cdot \log(|S|) + |A| \cdot \log(|S|)\Big)$ et donc cela donne une complexité en $O(|A| \cdot \log(|S|))$.

```
Procédure MaJ-F-Prim(F, d, G, s, Π, IndiceDansF)
//G = (S, A, w) : un graphe non-orienté, valué et connexe.
begin
    pour chaque (s, u) ∈ A faire
        si (IndiceDansF[u] ≠ −1) ∧ (w(s, u) < d[u]) alors
            Π(u) := s
            d[u] := w(s, u)
            //On réorganise...
            //F[i] désigne le sommet situé à la position i dans F.
            i := IndiceDansF[u]
            tant que (i/2 ≥ 1) ∧ (d[F[i/2]] > d[F[i]]) faire
                F[i] ↔ F[i/2]
                IndiceDansF[F[i]] := i
                IndiceDansF[F[i/2]] := i/2
                i := i/2;
end
```

Algorithme 17 : algorithme de Mise-à-Jour de la File F pour l'algorithme de Prim

8 Plus courts chemins

On considère le problème de la recherche de *plus courts chemins* dans un graphe orienté valué $G = (S, A, w)$ avec $w : A \to \mathbb{R}$.

Étant donné un chemin fini $\rho \stackrel{\text{def}}{=} v_0 \to v_1 \to \ldots \to v_k$, on note $w(\rho)$ sa longueur, c'est-à-dire la somme $\sum_{i=1,\ldots,k} w(v_{i-1}, v_i)$.

Définition 10 *Un chemin ρ de u à v est appelé un* plus court chemin *(PCC) de u à v ssi, pour tout chemin π de u à v, on a $w(\pi) \geq w(\rho)$.*

Existence des PCC :

Propriété 25 *Étant donnés un graphe G, et deux sommets u et v, il existe un plus court chemin entre u et v ssi*

*a v est atteignable depuis u (*i.e. $\exists u \to^* v$*), et*

b il n'existe pas de cycle strictement négatif $c : z \to^* z$ et un chemin $u \to^* z \to^* v$.

Preuve :

- ($\neg 2 \Rightarrow \neg 1$) Si il n'existe pas de chemin entre u et v, il n'y a clairement pas de PCC. Étant donné un cycle $c : z \to^* z$ t.q. $w(c) < 0$ et t.q. $\exists u \to^* z \to^* u$, il est clair que tout chemin entre u et v admet un autre chemin strictement plus court...
- ($2 \Rightarrow 1$) Supposons que tout sommet atteignable le long d'un chemin de u à v n'admet que des cycles positifs ou nuls, alors la recherche des PCC entre u et v peut se limiter à l'ensemble des chemins simples : tout chemin admet un chemin simple de poids inférieur ou égal. Or le nombre de chemins simples est fini. Donc il existe un ou plusieurs chemins de poids minimal.

□

Dans la suite, on suppose qu'il n'existe pas de cycle strictement négatif dans le graphe G.

On définit $\delta(s,u)$ la distance d'un PCC de s à u de la manière suivante :

$$\delta(s,u) \stackrel{\text{def}}{=} \begin{cases} \min\{w(\rho) \mid s \to_\rho u\} & \text{si } \exists\, s \to^* u \\ \infty & \text{sinon} \end{cases}$$

Les PCC admettent la propriété fondamentale suivante :

Propriété 26 *Si $\rho : v_0 \to v_1 \to \ldots \to v_k$ est un plus court chemin entre v_0 et v_k, alors tout sous-chemin $v_i \to \ldots \to v_j$ (avec $0 \leq i < j \leq k$) de ρ est un PCC de v_i à v_j.*

Preuve : Si ce n'était pas le cas, alors remplacer $v_i \to \ldots \to v_j$ par un PCC entre v_i et v_j diminuerait le poids du chemin entre v_0 et v_k et contredirait l'hypothèse de départ sur ρ. □

Les familles de problèmes sur les PCC. On peut considérer plusieurs problèmes de recherche de plus courts chemins :

- les PCC à origine unique : On cherche tous les PCC depuis un sommet de départ s ;
- les PCC à destination unique : On cherche tous les PCC menant à un sommet d'arrivée s ; et
- les PCC pour toutes les paires de sommets de G.

Ici on va voir un algorithme pour le premier type de problème (algorithme de Dijkstra) et pour le troisième (algorithme de Floyd).

Comme pour les ACM, on ne va pas chercher à calculer seulement les coûts minimaux des chemins, on cherche aussi à construire ces chemins.

Pour les PCC à origine unique, on va construire une "arborescence des PCC" T, c'est à dire un arbre de racine s, contenant tous les sommets accessibles depuis s et tel que tout chemin de s à x dans T soit un PCC de s à x dans G.

Propriété 27 *Soient G un graphe orienté valué et sans cycle strictement négatif, et s un sommet de G. Alors G possède une arborescence des PCC de racine s.*

Preuve : Soit S_{acc} l'ensemble des états accessibles depuis s. On veut montrer l'existence d'une arborescence contenant S_{acc} et dont les chemins sont des PCC. On montre l'existence d'une arborescence *partielle* des PCC, de taille k (taille = nb de sommets accessibles dans l'arbre) pour tout k de 1 à $|S_{acc}|$ par induction sur k.

- $k = 1$: le sommet isolé s est bien une arborescence partielle de PCC.
- $k+1$ avec $k < |S_{acc}|$: Soit $T = (S_T, A_T)$ une arborescence partielle de taille k. Soit $x \in S_{acc} \backslash S_T$. Soit $\rho : s = v_0 \to \ldots \to v_l = x$ un PCC de s à x dans G. Soit v_{i+1} le premier sommet le long de ρ tel que $v_{i+1} \notin S_T$ (et $v_{i+1} \in S_{acc}$).
  On sait que le préfixe $v_0 \to \ldots \to v_{i+1}$ est un PCC de s à v_{i+1} (d'après la prop. 26). Donc $T' = (S_T \cup \{v_{i+1}\}, A_T \cup \{(v_i, v_{i+1})\})$ est une arborescence partielle des PCC de taille $k+1$.

□

8.1 Algorithme de Dijkstra

Ici on prend une fonction poids w à valeurs dans $\mathbb{R}_+$. G ne contient donc pas de cycle strictement négatif. **NB :** Cette restriction est nécessaire pour l'algorithme de Dijkstra mais d'autres algorithmes n'en ont pas besoin (par ex. Bellman-Ford).

L'algorithme 18 décrit l'algorithme de Dijkstra, il utilise une file de priorité F avec une fonction de mise à jour "étendue" de la file lorsque l'on diminue la priorité d'un élément (le tableau IndiceDansF sert à accéder efficacement à la position dans F d'un élément donné). **Sa structure est très proche de celle de l'algorithme de Prim, c'est aussi un algorithme glouton.**

Au cours de l'algorithme, on distingue
- les sommets contenus dans la file : ceux pour lesquels on ne connaît pas encore la longueur d'un PCC depuis s ; la valeur $\mathsf{d}[-]$ correspond alors à une surapproximation de cette longueur.
- et les sommets déjà extraits de la file : ce sont ceux pour lesquels on sait que $\mathsf{d}[-]$ est égal à la longueur d'un PCC depuis s.

Procédure `PCC-Dijkstra`(G, s)
`//`$G = (S, A, w)$ `: un graphe orienté, valué avec` $w : A \to \mathbb{R}_+$.
`//`$s \in S$ `: un sommet origine.`
begin
  pour chaque $u \in S$ **faire**
    $\Pi[u] := \mathsf{nil}$
    $\mathsf{d}[u] := \begin{cases} 0 & \text{si } u = s \\ \infty & \text{sinon} \end{cases}$
  $F := \mathsf{File}(S, d, \mathsf{IndiceDansF})$ `//Construit` F `et initialise` IndiceDansF
  tant que $F \neq \emptyset$ **faire**
    $u := \mathsf{Extraire\text{-}Min}(F)$
    $\mathsf{IndiceDansF}[u] := -1$
    pour chaque $(u, v) \in A$ **faire**
      si $d[v] > d[u] + w(u, v)$ **alors**
        $\mathsf{d}[v] := \mathsf{d}[u] + w(u, v)$
        $\Pi[v] := u$
        $\mathsf{MaJ\text{-}F\text{-}Dijkstra}(F, d, v, \mathsf{IndiceDansF})$
  return d, Π
end

Algorithme 18 : algorithme de Dijkstra

Au cours de l'algorithme, on note $\mathsf{d}^i(v)$ la valeur de $\mathsf{d}(v)$ au début de la i-ème itération (il y a $|S|$ itérations).

On a les deux premières propriétés suivantes :

Propriété 28
- *La valeur $d[-]$ des sommets extraits à chaque itération est croissant, c'est-à-dire si v_j (resp. v_k) dénote le sommet extrait à l'itération j (resp. k) avec $j < k$, on a : $\mathsf{d}^j[v_j] \leq \mathsf{d}^k[v_k]$.*
- *Lorsqu'un sommet est extrait de F, son coefficient $d[-]$ n'est plus jamais modifié par l'algorithme.*

Preuve :

- On montre que l'on a $\mathsf{d}^i[v_i] \leq \mathsf{d}^{i+1}[v_{i+1}]$ par induction sur le numéro d'itération. Au cours de la première itération, c'est s qui est extrait et la valeur $\mathsf{d}[s]$ est 0 : c'est bien la distance minimale (pas de poids négatif). A la $i+1$-ème itération, on distingue deux cas :
  - si $\mathsf{d}^i[v_{i+1}] = \mathsf{d}^{i+1}[v_{i+1}]$ (*i.e.* la valeur $\mathsf{d}[v_{i+1}]$ n'a pas été mise à jour au cours de l'itération i) alors le résultat est vrai car v_i a été extrait avant et donc $\mathsf{d}^i[v_i] \leq \mathsf{d}^i[v_{i+1}]$;
  - Sinon il a été mis à jour avec une instruction de la forme $\mathsf{d}[v_{i+1}] := \mathsf{d}[v_i] + w(v_i, v_{i+1})$ et le résultat est donc vrai.
- Après l'extraction d'un sommet v_i, tout essai de modification lors de l'itération $j > i$ est conditionné par $\mathsf{d}^j[v_j] + w(v_j, v_i) < \mathsf{d}^i[v_i]$ ce qui est faux d'après la propriété précédente.

□

On a de plus :

Propriété 29 *A tout moment de l'algorithme, on a* $\mathsf{d}[u] \geq \delta(s, u)$

(c'est vrai lors de l'initialisation et c'est clairement maintenu à chaque itération.)

On peut maintenant énoncer la correction de l'algorithme :

Théorème 6 (Correction de l'algorithme de Dijkstra) *Étant donné un graphe* $G = (S, A, w)$ *orienté et valué tel que* $w : A \to \mathbb{R}_+$*, l'algorithme de Dijkstra*

1. *termine,*
2. *à la fin, on a* $\mathsf{d}[u] = \delta(s, u)$ *pour tout sommet* $u \in S$*, et*
3. *pour tout sommet* $u \in S \backslash \{s\}$*, si* $\mathsf{d}[u] < \infty$*, alors il existe un PCC de* s *à* u *dont le dernier arc est* $(\Pi[u], u)$*.*

Preuve : La terminaison découle directement de la boucle principale : il y a exactement $|S|$ itérations (F est initialisée avec $|S|$ sommets et chaque itération en extrait exactement un).

Pour prouver le second point de la correction, il suffit de montrer la propriété $\mathsf{d}[u] = \delta(s, u)$ au moment de l'extraction de u (cf. la propriété 28). On le montre par induction sur le numéro d'itération $i = 1 \ldots |S|$:

- $i = 1$: le premier sommet à être extrait est s, et $\mathsf{d}[s] = 0 = \delta(s, s)$.
- $i + 1$: Soit u le sommet extrait à l'itération $i + 1$. On distingue deux cas :
  - $\mathsf{d}[u] = \infty$: aucun sommet v extrait précédemment avec $\mathsf{d}[v] < \infty$ n'a de successeur dans F. Les sommets restant dans F (incluant u) ne sont donc pas accessibles depuis s et leur d ne bougera plus (seuls des sommets avec un d à ∞ pourront être extraits). Et donc $\mathsf{d}[u] = \delta(s, u)$.
  - $\mathsf{d}[u] \in \mathbb{R}_+$: Soit $\Pi(u) = v$. On a alors $\mathsf{d}[u] = \mathsf{d}[v] + w(v, u)$ et par h.i. $\mathsf{d}[v] = \delta(s, v)$. Soit ρ un PCC de s à u.
    Si le prédécesseur de u le long de ρ a déjà été extrait de F, alors soit c'est v (et on a le résultat), soit c'est un autre sommet z et alors on a $\mathsf{d}[z] + w(z, u) = \mathsf{d}[u]$ (ça ne peut être "<" car sinon le prédécesseur selon Π aurait été z, ni ">" car sinon ρ ne serait pas un PCC car $\mathsf{d}[z] = \delta(s, z)$ puisque z a été extrait avant) et donc on a bien $\mathsf{d}[u] = \delta(s, u)$.
    Sinon soit y le premier sommet de ρ qui est encore dans F lors de l'extraction de u et

soit x son prédécesseur sur $\rho : s \to^* x \to y \to^* u$. Par h.i. on a $\mathsf{d}[x] = \delta(s,x)$ et l'arc (x,y) a déjà été examiné, donc $\mathsf{d}[y] = \mathsf{d}[x] + w(x,y) = \delta(s,y)$ lors de l'extraction de u. Puisque u est traité avant y, c'est que l'on a $\mathsf{d}[u] \leq \mathsf{d}[y]$, et comme les arcs ont des poids **positifs** on a $\delta(s,u) \geq \delta(s,y)$. On en déduit (avec la propriété 29) que lors de l'extraction de u on a :

$$\mathsf{d}[u] \leq \mathsf{d}[y] = \delta(s,y) \leq \delta(s,u) \leq \mathsf{d}[u]$$

Et donc $\mathsf{d}[u] = \delta(s,u)$.

Pour le troisième point, il suffit de noter que la valeur $\Pi[u]$ correspond au sommet qui a permis de déduire la valeur finale de $\mathsf{d}[u]$, c'est-à-dire $\delta(s,u)$. □

L'opération de mise à jour de la file de priorité F est assez classique :

Procédure MaJ-F-Dijkstra($F, d, v,$ IndiceDansF)
begin
  //$F[i]$ `désigne le sommet situé à la position` i `dans` F.
  $i :=$ IndiceDansF$[v]$
  tant que $(i/2 \geq 1) \wedge (d[F[i/2]] > d[F[i]])$ **faire**
    $F[i] \leftrightarrow F[i/2]$
    IndiceDansF$[F[i]] := i$
    IndiceDansF$[F[i/2]] := i/2$
    $i := i/2$;
end

Algorithme 19 : algorithme de Mise-à-Jour de la File F pour PCC-Dijkstra

La procédure MaJ($F, d, v,$ IndiceDansF) prend un temps en $O(\log(|F|))$.

L'algorithme PCC-Dijkstra prend un temps en $O\Big((|S|+|A|)\cdot\log(|S|)\Big)$, c'est à dire en $O(|A|\cdot\log(|S|))$ lorsqu'on suppose $|S| \leq |A|$.

La figure 47 représente un exemple d'application de l'algorithme de Dijkstra : sur la figure de droite, on a représenté les arcs de l'arborescence des PCC et à côté de chaque sommet les nombres "$\mathbf{d}(l)$" indiquent que la distance trouvée est $\mathbf{d}$ et que le sommet a été extrait à l'itération l.

8.2 Algorithme de Floyd-Warshall

Maintenant nous considérons la recherche des PCC entre tous les sommets : nous voulons une procédure qui calcule la distance $\delta(x,y)$ – *i.e.* la longueur d'un PCC entre x et y – pour toute paire de sommet $\langle x,y\rangle$.

On va utiliser une représentation matricielle d'un graphe valué $G = (S,A,w)$ avec $S = \{x_1, \dots x_n\}$ et $w : A \to \mathbb{R}$. On note $M = (\alpha_{ij})_{1\leq i,j\leq n}$ la matrice représentant G où α_{ij} décrit l'arc entre x_i et x_j :

$$\alpha_{ij} \stackrel{\text{def}}{=} \begin{cases} 0 & \text{si } i = j \\ w(x_i,x_j) & \text{si } i \neq j \text{ et } (x_i,x_j) \in A \\ \infty & \text{si } i \neq j \text{ et } (x_i,x_j) \notin A \end{cases}$$

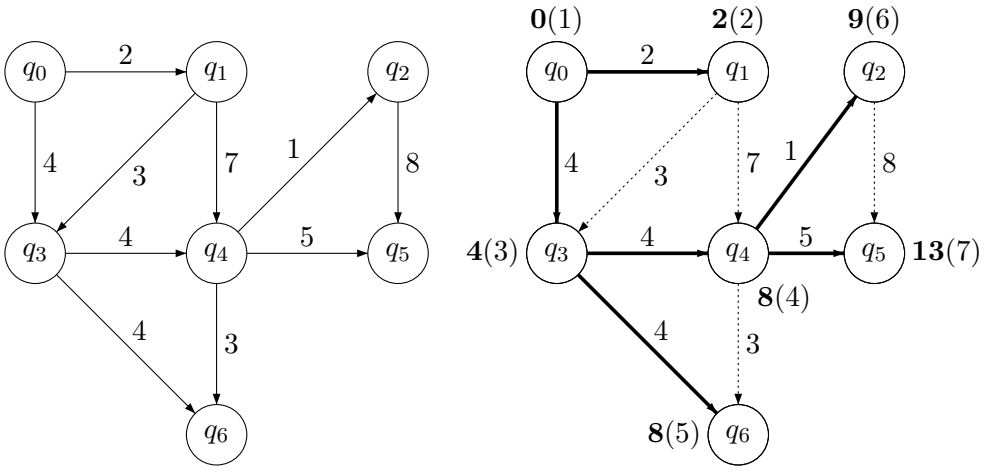

FIGURE 47 – Exemple d'application de PCC-Dijkstra avec depuis q_0.

Dans la suite, on notera δ_{ij} la distance $\delta(x_i, x_j)$.

Précédemment nous utilisions une fonction "prédécesseur", ici nous allons construire une matrice des prédécesseurs Π pour représenter **tous** les PCC : Il s'agit d'une matrice de taille $n \times n$, à coefficients π_{ij} dans S :

$$\pi_{ij} \stackrel{\text{def}}{=} \begin{cases} \mathsf{nil} & \text{si } i = j \text{ ou } x_i \not\rightarrow^* x_j \\ s & \text{si } s \text{ est le prédécesseur immédiat de } x_j \text{ le long d'un PCC entre } x_i \text{ et } x_j \end{cases}$$

L'algorithme de Floyd-Warshall est un algorithme de programmation dynamique qui permet de calculer les δ_{ij} et la matrice Π. Pour cela, on utilise une famille de matrices $D^{(k)}$ $(k = 1, \dots n)$ contenant les calculs intermédiaires, et la dernière $D^{(n)}$ contient les coefficients δ_{ij}. Il est décrit par l'algorithme 20.

La complexité (en temps) de cet algorithme est clairement en $O(n^3)$. Il utilise un espace mémoire en $O(n^3)$ mais il est possible de ramener cette complexité en $O(n^2)$(voir ci-dessous).

Étant donné un chemin $\rho : v_0 \to v_1 \to \ldots \to v_l$, on appelle l'intérieur de ρ l'ensemble des sommets $v_1, \dots, v_{k-1}$, c'est à dire l'ensemble des sommets traversés par ρ (hormis les extrémités) .

La correction de l'algorithme repose sur la propriété suivante :

Propriété 30 *Si G ne contient pas de cycle strictement négatif, alors pour tout $k = 0, \dots, n$, on a :*

- *la valeur calculée pour $d_{ij}^{(k)}$ correspond à la distance d'un PCC entre x_i et x_j et d'intérieur inclus dans $\{x_1, \dots, x_k\}$;*
- *$\pi_{ij}^{(k)}$ correspond au prédécesseur de x_j le long d'un PCC de x_i à x_j et d'intérieur inclus dans $\{x_1, \dots, x_k\}$.*

Preuve : On montre cela par induction sur k. Lorsque $k = 0$, les coefficients représentent les chemins élémentaires constitués d'une unique (ou de zéro) transition et donc d'intérieur vide.

Pour $k + 1$, considérons deux sommets x_i et x_j. Considérons un PCC simple (il en existe toujours car il n'y a pas de cycle négatif) d'intérieur $\{x_1, \dots, x_{k+1}\}$ passant par x_{k+1}, alors il

Procédure `PCC-Floyd`(G)
`//`$G = (S, A, w)$ `: un graphe orienté, valué avec` $w : A \to \mathbb{R}$.
`//avec` $S = \{x_1, \ldots, x_n\}$
`//avec` $M = (\alpha_{ij})_{1 \leq i,j \leq n}$ `la matrice corresp. à` A
begin
  `//On initialise` $D^{(0)}$ `avec` M`:`
  pour $i = 1 \ldots n$ **faire**
    pour $j = 1 \ldots n$ **faire**
      $d_{ij}^{(0)} := \alpha_{ij}$
      si $\alpha_{ij} \neq \infty$ **alors** $\pi_{ij}^{(0)} := i$
  pour $k = 1 \ldots n$ **faire**
    pour $i = 1 \ldots n$ **faire**
      pour $j = 1 \ldots n$ **faire**
        si $d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ **alors**
          $d_{ij}^{(k)} := d_{ij}^{(k-1)}$
          $\pi_{ij}^{(k)} := \pi_{ij}^{(k-1)}$
        sinon
          $d_{ij}^{(k)} := d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$
          $\pi_{ij}^{(k)} := \pi_{kj}^{(k-1)}$
  return D, Π
end

Algorithme 20 : algorithme de Floyd-Warshall

est composé d'un PCC entre x_i et x_{k+1} et d'un PCC entre x_{k+1} et x_j (cf la structure générale des PCC) et ils sont chacun d'intérieur inclus dans $\{x_1, \ldots, x_k\}$, on obtient donc leur distance avec $d_{ik}^{(k)}$ et $d_{kj}^{(k)}$ par hypothèse d'induction.
Si il existe un PCC d'intérieur $\{x_1, \ldots, x_k\}$ de même poids que les PCC d'intérieur $\{x_1, \ldots, x_{k+1}\}$, alors le coefficient ne change pas et l'hypothèse d'induction suffit à conclure.

Pour la matrice des prédécesseurs, le même principe s'applique. □

On en déduit le théorème suivant :

Théorème 7 *Si G ne contient pas de cycle strictement négatif, alors $D^{(n)}$ contient les coefficients δ_{ij} des PCC.*

La preuve est directe : sans cycle négatif, il existe toujours des PCC simples d'intérieur $\{x_1, \ldots, x_n\}$, et on obtient leur distance après la n-ième itération.

L'algorithme de Floyd-Warshall permet aussi de détecter la présence de cycle strictement négatif :

Propriété 31 *G contient un cycle strictement négatif ssi il existe un coefficient $d_{ii}^{(n)}$ strictement négatif.*

Preuve : (1) $\Rightarrow$ (2) : Soit c un cycle $x_i \to \ldots \to x_i$ de poids strictement négatif et d'intérieur inclus dans $\{x_1, \ldots, x_k\}$ tel que x_k apparaisse effectivement le long de c ($k \neq i, j$).

La présence des cycles négatifs fait que les valeurs $d_{ij}^{(l)}$ calculées par l'algorithme correspondent à des longueurs de chemins qui ne sont pas toujours des chemins simples : pour aller de x_i à x_j, il peut être plus court de passer par x_1 plusieurs fois... Ce que l'on sait, c'est que les distances calculée sont inférieures ou égale à celles correspondant aux chemins simples entre x_i et x_j. Lorsqu'on calcule $d_{ii}^{(k)}$ en comparant $d_{ii}^{(k-1)}$ et $d_{ik}^{(k-1)} + d_{ki}^{(k-1)}$, on obtient un nombre strictement négatif et la valeur de $d_{ii}^{(n)}$ le sera donc encore à la fin de l'algorithme.

(2) $\Rightarrow$ (1) : Tout coefficient correspond à la longueur d'un certain chemin (pas toujours simple !). Un coefficient négatif sur la diagonale correspond donc à une boucle négative. □

L'algorithme de Floyd-Warshall permet donc de calculer tous les PCC en cas d'absence de cycle négatif et de tester la présence de tels cycles.

On peut simplifier l'algorithme précédent en ne considérant qu'une seule matrice de calcul D comme c'est fait dans l'algorithme 21.

Procédure `PCC-Floyd`(G)
`//`$G = (S, A, w)$ `: un graphe orienté, valué avec` $w : A \to \mathbb{R}$.
`//avec` $S = \{x_1, \ldots, x_n\}$
`//avec` $M = (\alpha_{ij})_{1 \leq i,j \leq n}$ `la matrice corresp. à` A
begin
  `//On initialise` D `avec` M`:`
  pour $i = 1 \ldots n$ **faire**
    pour $j = 1 \ldots n$ **faire**
      $d_{ij} := \alpha_{ij}$
      si $\alpha_{ij} \neq \infty$ **alors** $\pi_{ij} := i$
  pour $k = 1 \ldots n$ **faire**
    pour $i = 1 \ldots n$ **faire**
      pour $j = 1 \ldots n$ **faire**
        si $d_{ij} > d_{ik} + d_{kj}$ **alors**
          $d_{ij} := d_{ik} + d_{kj}$
          $\pi_{ij} := \pi_{kj}$
  return D, Π
end

Algorithme 21 : algorithme de Floyd-Warshall (version améliorée)

Propriété 32 *L'algorithme "amélioré" est correct :*
- *il calcule les mêmes coefficients que la première version lorsqu'il n'y a pas de cycle négatif ;*
- *il détecte les cycles négatifs comme précédemment.*

Preuve : En cas d'absence de cycle négatif : le coefficient d_{ij} après l'itération k avec l'algorithme amélioré est égal au coefficient $d_{ij}^{(k)}$. En effet : il dépend de $d_{ij}^{(k-1)}$, $d_{ik}^{(k-1)}$ et $d_{kj}^{(k-1)}$: le premier est ok par h.i. (les première modifications liées à l'itération k ne l'ont pas modifié),

et les deux autres non plus car sans cycle négatif, $d_{kk}^{(l)} \geq 0$ et donc il n'est jamais intéressant de traverser x_k pour aller de x_i à x_k (ou pour aller de x_k à x_j).

Si il y a des cycles négatifs, on va les détecter comme précédemment car les coefficients d_{ij} correspondent toujours à des distances de chemins. □

Finalement, on peut utiliser la matrice Π pour retrouver un PCC entre deux sommets avec l'algorithme 22.

Procédure `Construire-PCC`(Π, i, j)
`//`Π `: une matrice de prédécesseurs.`
`//`$1 \leq i, j \leq n$ `: deux indices de sommets`
begin
  si $i \neq j$ **alors**
    return *Construire-PCC*$(\Pi, i, \Pi[i, j]) \oplus (\Pi[i, j], j)$
end

Algorithme 22 : Construction du chemin à partir de Π.