Algorithmique 2

2020-2021

Table des matières

Chapitre 1

Rappels

1.1 Notations

Les algorithmes seront décrits dans un pseudo-code suffisamment lisible pour ne pas laisser d'ambiguïté. Nous utiliserons uniquement les constructions et les notations les plus standards. L'affectation d'une valeur v à une adresse dénotée par une référence (un pointeur) r sera dénotée par $r \leftarrow v$. La valeur contenue dans une référence r sera dénoté $!\mathsf{r}$. Le test d'égalité est bien sûr noté $=$. L'échange des contenus des cases i et j d'un tableau t est noté $\mathsf{t}.[i] \leftrightarrow \mathsf{t}.[j]$. Les fonctions sont définis selon la syntaxe :

fonction nom_de_fonction($type\_argument\_1, \ldots$) : $type\_resultat =$ corps de la fonction

La définition d'une variable est introduite par le mot clé **soit**, le symbole d'initialisation est $:=$. Les mot-clés du langage sont écrits en gras, les types en italique, et les identifiants (noms de variables et de fonctions) avec une police sans serif, tel que ci-dessus. Nous utiliserons autant que possible les notations standards mathématiques pour tous les opérateurs mathématiques, par exemple pour les opérateurs booléens $(\wedge, \vee, \neg)$.

Nous spécifierons la sémantique des algorithmes en utilisant aussi les notations mathématiques standards. Une sémantique mathématique précise sera donnée à chaque type de donnée. Les types entiers, booléens, flottants ont pour sémantiques respectives les entiers, les valeurs de vérités (vrai et faux), et les réels (approximation qui nous suffira). Les tableaux de longueur l ont pour sémantique les fonctions sur le domaine $[0, l-1]$. La sémantique d'une expression expr sera dénotée par $[\![\text{expr}]\!]$. Cette sémantique comporte deux aspects : les effets de bord (principalement les altérations de la mémoire), et la valeur retournée par l'évaluation de l'expression. Nous noterons alors $[\![\text{expr}]\!] =$ effets de bord; résultat retourné. Si l'un des deux termes est vide, nous n'écrivons que l'autre. En plus des notions standards de mathématiques, afin de supporter le modèle RAM, nous devons définir les notion de références et d'affectation, qui ne sont pas *stricto sensu* des opérations mathématiques. Le modèle RAM (Random Access Memory) suppose l'existence d'une notion de temps et de mémoire, que nous gardons implicites. Nous noterons alors $\leftarrow$ l'affectation d'une valeur à une adresse de cette mémoire, de sorte que $[\![\mathsf{r} \leftarrow \mathsf{v}]\!] = [\![\mathsf{r}]\!] \leftarrow [\![\mathsf{v}]\!]$, et **ref**$(x)$ l'adresse d'un espace mémoire contenant x. $!a$ correspond au contenu de l'espace mémoire d'adresse a. Nous préciserons suffisamment le pseudocode pour éviter les ambigüités sur l'ordre d'évaluation, la seule exception concerne les opérateurs booléens qui sont évalués paresseusement de gauche à droite.

1.2 Structures de données

Les structures de données permettent d'enregistrer et de récupérer des valeurs en cours de calculs, au travers d'une interface simple. Les structures ainsi définies peuvent être mise à contribution dans des algorithmes plus complexes, en faisant abstraction de leur implémentation. Nous faisons donc la distinction entre structures de données *abstraite* et *concrète*.

type *liste* **de** t	séquence finie d'éléments de type t
liste_vide : *liste*	$[\![\text{liste\_vide}]\!] = \langle\rangle$
est_vide(*list*) : *bool*	$[\![\text{est\_vide(l)}]\!] =$ vrai si $[\![l]\!] = \langle\rangle$ $[\![\text{est\_vide(l)}]\!] =$ faux sinon
insére(t, *liste*) : *liste*	$[\![\text{insére(elt, l)}]\!] = \langle [\![\text{elt}]\!], e_k, e_{k-1}, \ldots, e_0\rangle$, avec $[\![\text{l}]\!] = \langle e_k, \ldots, e_0\rangle$
tete(*liste*) : t	$[\![\text{tete(l)}]\!] = e_k$ lorsque $[\![\text{l}]\!] = \langle e_k, \ldots, e_0\rangle$, Précondition : $[\![\text{l}]\!] \neq \langle\rangle$
queue(*liste*) : *liste*	$[\![\text{queue(l)}]\!] = \langle e_{k-1}, \ldots, e_1, e_0\rangle$ lorsque $[\![\text{l}]\!] = \langle e_k, \ldots, e_0\rangle$, Précondition : $[\![\text{l}]\!] \neq \langle\rangle$

FIGURE 1.1 – La structure de données abstraite *liste*

Définition 1.2.1. *Une* structure de données abstraite *est la spécification mathématique de types de données et d'opérations sur ces types.*

En général, une structure de données abstraite spécifie un type de donnée particulier et des opérations visant à manipuler ce type de donnée. Une structure de données abstraite ne précise pas comment ces opérations peuvent être implantées dans un langage de programmation. Il s'agit exclusivement d'une interface. Une même structure de données abstraite peut être implantée par plusieurs algorithmes différents. Chaque implantation est alors une structure de donnée concrète.

Définition 1.2.2. *Une* structure de données concrète *(ou* structure de donnée*) est une description de l'implantation d'une structure de données abstraite dans un modèle de calcul précis. Dans le cadre de ce cours, cette description est un pseudocode pour le modèle RAM.*

Exemple 1.2.1 (Les structures linéaires : les listes). Les listes sont une structure de données abstraite pour représenter les séquences, avec en accès exclusivement au dernier élément ajouté dans la séquence (politique du *dernier entré premier sorti*, dit LIFO). Contrairement aux piles, les listes sont persistantes : les fonctions d'insertion et du suppression ne modifie pas la liste passée en argument mais retourne une liste fraîche.

La Figure 1.1 donne la spécification de cette structure de données abstraite. Nous donnons comme sémantique aux listes les séquences d'entiers, notées entre $\langle \ldots \rangle$. Nous pouvons donc préciser formellement ce que nous attendons des différentes opérations sur les listes grâce à nos notations.

Exemple 1.2.2 (Les structures linéraires : les piles). Les piles sont aussi une structure de données abstraite qui permet de représenter les séquences avec une politique LIFO. Contrairement aux listes, une pile est modifiée par l'appel des opérations de pile, comme le reflète les types des fonctions de la Figure 1.2. Celle-ci détaille la spécification de cette structure de données abstraite.

On note que la différence avec les listes est assez minime, et souvent les deux structures de données abstraites seront interchangeables. Les listes ont cependant l'avantage d'être persistantes : on ne peut pas les détruire, en revanche, cela peut avoir un coup non-négligeable en mémoire si elle sont implantées dans des langages sans désallocation automatique de la mémoire. Nous n'explorerons cependant pas plus ces subtilités.

Exemple 1.2.3 (Les listes simplement chainées). Les *listes comme liste simplement chainée* et *piles comme liste simplement chainée* sont deux structures de données *concrètes*, qui permettent d'implanter respectivement les listes ou les piles à partir d'une même idée. Notons $e_k, \ldots, e_1, e_0$ la séquence encodée. Une liste simplement chainée est formée d'un ensemble indicé de *maillons* $m_k, \ldots, m_0$ tel que chaque maillon m_i contient une paire, dont le premier élément est e_i et le second élément est m_{i-1} si $i \neq 0$, ou une valeur fixe notée $\perp$ sinon. La liste simplement chainée est ensuite manipulée à partir du maillon de tête m_k.

Ainsi, les piles comme listes simplement chainées sont détaillés en Figure 1.3.

type *pile* **de** t	référence à une séquence finie d'éléments de type t
pile_vide() : *pile*	$[\![\text{pile\_vide}()]\!] = \textbf{ref}\langle\rangle$
est_vide(*pile*) : *bool*	$[\![\text{est\_vide}(\text{p})]\!]$ = vrai si $![\![p]\!] = \langle\rangle$, $[\![\text{est\_vide}(\text{p})]\!]$ = faux sinon
empile(t, *pile*) : *void*	$[\![\text{empile}(\text{elt}, \text{p})]\!] = [\![\text{p}]\!] \leftarrow \langle [\![\text{elt}]\!], e_k, e_{k-1}, \ldots, e_0\rangle$, avec $![\![\text{p}]\!] = \langle e_k, \ldots, e_0\rangle$
sommet(*pile*) : t	$[\![\text{sommet}(\text{p})]\!] = e_k$ lorsque $![\![\text{p}]\!] = \langle e_k, \ldots, e_0\rangle$, Précondition : $![\![\text{p}]\!] \neq \langle\rangle$
dépile(*pile*) : *void*	$[\![\text{dépile}(\text{p})]\!] = [\![\text{p}]\!] \leftarrow \langle e_{k-1}, \ldots, e_1, e_0\rangle$ lorsque $![\![\text{p}]\!] = \langle e_k, \ldots, e_0\rangle$, Précondition : $![\![\text{p}]\!] \neq \langle\rangle$

FIGURE 1.2 – La structure de données abstraite *pile*

```
1   type maillon de t =
2     struct{contenu : t; suivant : maillon}
3     ou bien ⊥
4
5   type pile de t = ref maillon
6
7   fonction pile_vide() : pile = retourner ref(⊥)
8
9   fonction est_vide(pile p) : bool = retourner !p = ⊥
10
11  fonction empile(t elt, pile p) = p ← {contenu : elt; suivant : !p}
12
13  fonction sommet(pile p) : t = retourner !p.suivant
14
15  fonction depile(pile p) = p ←!p.suivant
```

FIGURE 1.3 – Pseudocode pour les piles codées par des listes simplement chainées

Ce pseudocode suggère plusieurs remarques. Selon les langages de programmation choisis, des modifications devront être apportées à cette base en vue d'une implantation. Par exemple :

— le type *maillon* est défini comme une disjonction (**ou bien**). Certains langages ne supportent peu ou pas cette construction,
— le type du champ suivant peut aussi poser problème dans les langages de bas niveau. On peut alors le remplacer par le type **ref** maillon, en adaptant le reste du code,
— dans les langages qui requièrent une désallocation explicite de la mémoire, depile doit désallouer l'espace mémoire contenant le maillon référencé dans p avant l'affectation,
— lorsqu'une précondition est présente, on devrait la tester et ajouter la levée d'une exception.

Tous ces détails sont de l'ordre de la programmation et non de l'algorithmique, nous ne mentionnerons donc plus ces difficultés qui n'entrent pas dans le cadre de ce cours.

Les piles et les listes ne diffèrent que par des détails, comme le révèle leurs interfaces respectives Figure 1.2 et Figure 1.1. En fait, elles peuvent être vues comme des variantes l'une de l'autre. Les piles sont une structure destructive : les opérations d'insertion et de suppression modifient l'état de la mémoire, de sorte que seule la nouvelle version de la pile survit. Au contraire, les listes sont une version persistante : l'insertion et la suppression d'éléments créent une nouvelle liste, comme le montre le type de retour des deux fonctions. Ainsi, la liste avant insertion ou suppression continue d'exister et d'être une instance valide de liste, en parallèle avec la nouvelle instance créée lors de l'appel de fonction.

Les structures de données persistantes ont cet avantage de ne jamais être modifiées ou détruites qui peut être utile pour écrire des algorithmes. En échange, la persistance peut avoir un coût sur l'utilisation de la mémoire, et dans certains cas les meilleurs algorithmes persistants sont moins efficaces que leurs contreparties destructives.

Les structures de données abstraites que nous serons amenées à étudier sont parfois persistantes, parfois destructives. La signature des principales fonctions permet de repérer le caractère persistant ou destructif de la structure. Dans le cadre de ce cours, nous n'utiliserons jamais la persistance *per se*, et donc cette distinction entre persistance et destructivité ne sera pas mise en valeur, mais explique certaines différences entre les structures de données abstraites et concrètes qui seront données.

Exemple 1.2.4 (Les files de priorité (ou tas)). Les *files de priorités* (aussi appelés tas, anglicisme pour le terme *heap* utilisé en anglais) sont une structure de données abstraite, codant des ensembles d'éléments ordonnés, avec les opérations d'insertion, de recherche du minimum et de suppression du minimum. La spécification des files de priorités est donnée par la Figure 1.6.

Les tas binaires sont un exemple de structure de données concrète implantant les files de priorités. Les tas binaires ont été étudiées en cours d'algorithmique I (dans le cadre du tri par tas). Pour rappel, un tas binaire est un arbre binaire complet, chaque nœud contenant un élément du tas de type t, avec comme invariant :

$$\mathsf{n.pere.contenu} \leq \mathsf{n.contenu} \qquad \text{(pour tout nœud n distinct de la racine)}$$

Ainsi, l'élément minimum du tas est contenu par la racine. L'arbre binaire est encodé comme un tableau, avec la racine à l'indice 0. Le fils gauche d'un nœud d'indice i a alors pour indice $2 \cdot i + 1$, et le fils droit $2 \cdot i + 2$ (*cf.* Figure 1.4). Un exemple de représentation de tas est donné en Figure 1.5, avec l'arbre binaire correspondant.

Nous étudierons d'autres structures de données concrètes pour les files de priorités, comme alternatives aux tas binaires.

Pour finir, nous donnons une description des tas binaires en Figure 1.7. Pour rester simple, nous supposons que nous disposons d'une implémentation de la structure de données abstraite *tableau dynamique*, qui sont des tableaux dont la taille est variable et augmente automatiquement en fonction des besoins.

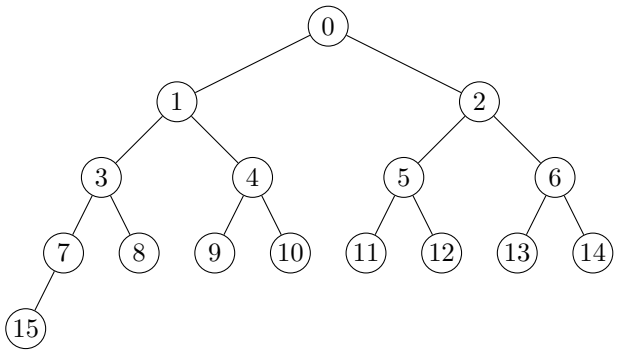
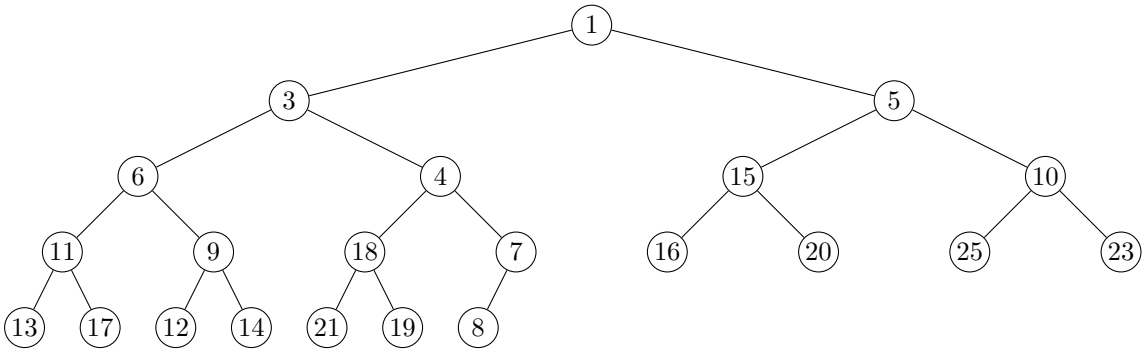


FIGURE 1.4 – Indexation des nœuds d'un tas binaire pour la représentation par tableau.



1	3	5	6	4	15	10	11	9	18	7	16	20	25	23	13	17	12	14	21	19	8

FIGURE 1.5 – Un tas binaire (tableau) et sa représentation comme arbre binaire complet.

type *tas* **de** t **avec fonction** $\leq(t,t)$: *bool*	référence sur un multi-ensemble d'éléments de type t
tas_vide() : *tas*	$[\![\text{tas\_vide}()]\!] = \textbf{ref}(\varnothing)$
est_vide(*tas*) : *bool*	$[\![\text{est\_vide(h)}]\!]$ = vrai si $![\![\text{h}]\!] = \varnothing$, $[\![\text{est\_vide(h)}]\!]$ = faux sinon
insére(t, *tas*) : *void*	$[\![\text{insére(elt, h)}]\!] = [\![h]\!] \leftarrow \{\text{elt}\} \cup [\![!\text{h}]\!]$
minimum(*tas*) : t	$[\![\text{minimum(h)}]\!] = \min(![\![\text{h}]\!])$, Précondition $![\![\text{h}]\!] \neq \varnothing$
extrais_min(*tas*) : t	$[\![\text{extrais\_min(h)}]\!] = [\![\text{h}]\!] \leftarrow ![\![\text{h}]\!] \setminus \{\min(![\![\text{h}]\!])\}; \min(![\![\text{h}]\!])$, Précondition : $![\![\text{l}]\!] \neq \varnothing$

FIGURE 1.6 – La structure de données abstraite *file de priorité* (variante destructive)

```
1   type tas de t avec fonction ≤(t, t) : bool =
2     Tableau Dynamique de t
3
4   fonction tas_vide : tas = retourner []   // le tableau dynamique vide
5
6   fonction est_vide(tas h) : bool = retourner longueur(h) = 0
7
8   fonction minimum(tas h) : t =
9     retourner tas.[0]
10
11  fonction pere(entier indice) : entier = retourner (indice − 1)/2
12  fonction fils_droit(entier indice) : entier = retourner 2 × indice + 2
13  fonction fils_gauche(entier indice) : entier = retourner 2 × indice + 1
14
15  fonction remonte(tas h, entier noeud) =
16    si noeud ≠ 0 ∧ h.[noeud] < h.[pere(noeud)] alors
17      h.[noeud] ↔ h.[pere(noeud)];
18      remonte(pere(noeud))
19
20  fonction descend(tas h, entier noeud) =   // on suppose h.[i] = +∞ si h.(i) est indéfini
21    si h.[noeud] > min(h.[fils_gauche(noeud)], h.[fils_droit(noeud)]) alors
22      si h.[fils_gauche(noeud)] < h.[fils_droit(noeud)] alors
23        h.[fils_gauche(noeud)] ↔ h.[noeud];
24        descend(fils_gauche(noeud))
25      sinon
26        h.[fils_droit(noeud)] ↔ h.[noeud];
27        descend(fils_droit(noeud))
28
29  fonction insérer(tas h, t elt) =
30    soit l := longueur(h);
31    h.[l] ← elt;   // h grandit d'un élément
32    remonte(h, l)
33
34  fonction extraire_min(tas h) =
35    soit l := longueur(h) − 1;
36    h.[0] ↔ h.[l];
37    supprimer l'indice l de h;
38    descend(h, 0)
```

FIGURE 1.7 – La structure de données concrète *tas binaire*

1.3 Preuves de correction

La moindre des qualités que nous puissions demander à un algorithme est qu'il soit correct. La correction repose sur deux propriétés : l'algorithme doit retourner une solution en temps fini (*terminaison*), et la solution doit être conforme à la spécification de l'algorithme (ou de la structure de données abstraite dans le cas de l'implémentation d'une structure de données concrète). En règle générale, la terminaison sera démontrée comme une conséquence de l'analyse de complexité asymptotique.

La conformité aux spécifications est prouvé à l'aide d'invariants.

Définition 1.3.1. *Un* invariant d'un type de données *est une propriété que vérifie toute instance correcte de ce type de données.*

Il peut arriver lors de l'exécution d'un algorithme qu'un invariant de type soit violé : l'instance devient alors incorrecte, et il est nécessaire de démontrer que l'instance sera corrigée d'ici la fin de l'exécution de l'algorithme.

Définition 1.3.2. *Un* invariant d'algorithme *est une propriété que vérifie les valeurs manipulées par un algorithme à tout instant de son exécution.*

Une algorithme est alors prouvé en supposant que les invariants de types de données définis sont vrais sur les données de l'algorithme, et en démontrant qu'ils sont conservés à la fin de l'exécution de l'algorithme et que le résultat est conforme aux spécifications. Pour cela, il peut être nécessaire d'utiliser des invariants d'algorithme en démontrant qu'ils sont vrais au début de l'exécution, et reste vrai à chaque étape du calcul.

Certains algorithmes ne sont corrects que si les données vérifient des propriétés additionnelles. D'autres assurent des propriétés plus fortes sur le résultat de leur exécution, ces propriétés pouvant être par la suite utilisées dans la preuve de correction d'un autre algorithme.

Définition 1.3.3. *Une* précondition *d'un algorithme est une propriété sur les données de l'algorithme, nécessaire à la correction de cet algorithme.*

Définition 1.3.4. *Une* postcondition *d'un algorithme est une propriété sur le résultat de l'exécution d'un algorithme, ou sur l'état des données à la fin de l'exécution d'un algorithme.*

Dans le cas d'algorithme récursif, une preuve par induction sera en plus nécessaire pour traiter des appels récursifs. Puisque nous sommes dans le domaine de la preuve, il n'existe pas de solutions générales pour démontrer la correction d'un algorithme. L'apprentissage de ces techniques se fera donc pour l'essentiel grâce aux exemples qui jalonnent le cours. Nous en examinons un immédiatement.

Exemple 1.3.1 (tas binaire (suite)). Nous avons déjà décrit les invariants des tas binaires informellement. Nous reprenons plus formellement.

Soit h un tas contenant les éléments $[\![\mathsf{h}]\!] = H$, alors h doit vérifier les invariants du type des tas, que nous introduisons ici. Nous rappelons que les tas s'interprètent sémantiquement comme des multi-ensembles d'éléments ordonnés (*cf.* Figure 1.6). Les invariants sont :

$$[\![\mathsf{longueur}(\mathsf{h})]\!] = |H| \tag{1.1}$$

$$H = \{[\![\mathsf{h}.[i]]\!] \ : \ i \in [0, |H| - 1]\} \tag{1.2}$$

$$\text{pour tout } i \in [1, |H| - 1], [\![\mathsf{h}.[\mathsf{pere}(i)]]\!] \leq [\![\mathsf{h}.[i]]\!] \tag{1.3}$$

(1.1) assure que le tableau est de même taille que le tas, (1.2) certifie que ce tableau contient bien tous les éléments du tas précisement. (1.3) garantie que le tas est correctement ordonné, le père est plus petit que le fils, et donc l'élément le plus petit est en racine.

À titre d'exemple, nous allons (seulement) prouver l'implémentation de la fonction insère de la Figure 1.7. Pour cela, nous aurons besoin d'une précondition pour la fonction remonte, qui est une relaxation de l'invariant (1.3). On définit comme précondition de $\mathsf{remonte}(\mathsf{h}, \mathsf{noeud})$:

$$\text{pour tout } i \in [1, |H| - 1] \setminus [\![\mathsf{noeud}]\!], [\![\mathsf{h}.[\mathsf{pere}(i)]]\!] \leq [\![\mathsf{h}.[i]]\!] \tag{1.4}$$

Démonstration. Prouvons d'abord la fonction remonte, en supposant les invariants de types (1.1) et (1.2) et la précondition (1.4) pour les arguments h et noeud, et en prouvant que h est une instance correcte de tas à la fin de l'exécution de remonte. Nous procédons par récurrence sur $[\![\mathsf{noeud}]\!]$.

Si $[\![\mathsf{noeud}]\!] = 0$, (1.4) s'instancie exactement en (1.3). Comme nous avons supposé les deux autres invariants satisfaits, et que h n'est pas modifié, h est bien un tas correct à la fin de l'exécution de remonte.

Si $i := [\![\mathsf{noeud}]\!] \neq 0$ et que avant l'exécution de remonte, $[\![\mathsf{h}.[i]]\!] > [\![\mathsf{h}.(\mathsf{pere}(i))]\!]$, alors par la précondition (1.4), l'invariant (1.3) est satisfait. À nouveau l'algorithme est correct.

Enfin, si $i := [\![\mathsf{noeud}]\!] \neq 0$ et que initialement $[\![\mathsf{h}.[i]]\!] \leq [\![\mathsf{h}.[\mathsf{pere}(i)]]\!]$, après l'exécution de la ligne 17 on a $[\![\mathsf{h}.[i]]\!] \geq [\![\mathsf{h}.(\mathsf{pere}(i))]\!]$. De plus cette ligne préserve les invariants (1.1) et (1.2), puisqu'on échange seulement la position de deux éléments. Par la précondition (1.4), comme seuls les nœuds d'indice i et $[\![\mathsf{pere}(i)]\!]$ sont modifiés, la précondition (1.4) pour $\mathsf{remonte}(\mathsf{h}, \mathsf{pere}(i))$ est satisfaite. Par hypothèse de récurrence, comme $[\![\mathsf{pere}(i)]\!] < i$, h est bien une instance correcte de tas à la fin de l'exécution de la fonction remonte.

Ceci termine la preuve de correction de remonte. La preuve de correction de insère est maintenant assez simple. On suppose les invariants de tas satisfaits pour h avant l'exécution de l'algorithme, et nous devons prouver que les invariants sont toujours satisfaits à la fin de son exécution. De plus il faut vérifier que la sémantique de l'opération insère, telle que définit par la structure de données abstraire 1.6, est respectée.

L'insertion d'un élément à l'indice l garantie que la longueur du tableau augmente de 1, donc (1.1) est satisfait avant l'appel à la fonction remonte. Puisque l'élément inséré est elt, et selon la sémantique de l'opération d'insertion dans les tas, (1.2) est aussi vrai. Enfin, tout noeud présent dans h au début de l'exécution de insère garde le même contenu ainsi que son père, donc (1.3) ne peut être violée que par le nouveau nœud d'indice l, donc la précondition (1.4) est vérifiée pour l'appel de remonte à la ligne 32. L'instance h vérifie donc les invariants de type à la fin de l'exécution de l'algorithme, et la sémantique de l'insertion est respectée, l'algorithme est donc correct. □

1.4 Analyse asymptotique

Parmi les nombreuses façons de mesurer les performances d'un algorithme, nous utiliserons la plus populaire (sauf mention contraire) : la complexité asymptotique dans le pire des cas, mesurée en nombre d'opérations élémentaires. Nous gardons la notion d'opérations élémentaires volontairement floue ; en fonction des problèmes que nous voudrons résoudre, nous pourrons utiliser différentes définitions, afin de modéliser au mieux les performances réelles de l'algorithme.

En règle générale, les opérations élémentaires sont celle du modèle RAM : exécution d'un instruction, allocation ou lecture d'un bloc mémoire unitaire... Sauf cas particulier, les opérations arithmétiques seront supposées élémentaires. Il s'agit d'une approximation correcte tant que les entiers manipulés sont de tailles bornées (au plus 2^{64} par exemple), ce qui est le plus souvent le cas en pratique, et que les flottants assurent une précision suffisante pour le calcul (ce qui est moins souvent le cas, mais nous ne verrons pas d'algorithme pour lequel la précision de l'arithmétique des flottants joue un rôle).

Enfin la complexité est mesurée le plus souvent en fonction de la taille d'un encodage compact de l'entrée de l'algorithme. Nous ne définirons pas formellement ces encodages, et la taille d'une entrée sera toujours dans notre cas une quantité facile à estimer que nous préciserons au cas par cas.

Nous rappelons les notations asymptotiques, dites de Landau :

Définition 1.4.1. *Soit $f\colon \mathbb{N}^* \to \mathbb{N}^*$, $g\colon \mathbb{N}^* \to \mathbb{N}^*$ deux fonctions, on dit que :*

- *f est négligeable devant g si pour tout $\varepsilon > 0$, il existe $N \in \mathbb{N}^*$ tel que pour tout $n \geq N$, $f(n) \leq \varepsilon \cdot g(n)$, et on le note $f(n) = o(g(n))$,*
- *f est dominée par g s'il existe $M > 0$, et $N \in \mathbb{N}^*$ tels que pour tout $n \geq N$, $f(n) \leq M \cdot g(n)$, et on le note $f(n) = O(g(n))$, on note aussi $g(n) = \Omega(f(n))$,*

— *f et g sont équivalentes si f et g sont non nulles à partir d'un certain rang et* $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$ *; on le note* $f \sim g$.

Exemple 1.4.1 (Tas binaire (suite)). Nous analysons maintenant la complexité asymptotique de l'opération d'insertion dans un tas binaire. insère fait appel à remonte, et ces deux fonctions utilisent en dehors des appels récursifs, uniquement un nombre constant k d'opérations élémentaires. La complexité pour un tas $[\![\mathsf{h}]\!] = H$ avec $|H| = n$ éléments, est donc au plus $k \times (p+1)$, où p est le nombre d'appels récursifs à la fonction remonte. Nous bornons p.

Chaque appel récursif de remonte divise par 2 l'indice du nœud passé en argument. En notant l'indice i_k au k^{e} appel :

$$0 \leq i_{k+1} \leq \frac{i_k}{2}$$

La première inégalité provenant de la condition d'arrêt de la fonction. Un simple raisonnement par récurrence donne alors $1 \leq i_{p-1} \leq \frac{i_0}{2^p}$, et comme $i_0 = n$, $2^p \leq n$, puis par croissance du logarithme en base 2, $p \leq \log n$.

On en déduit que la complexité asymptotique de insère est $O(\log n)$ sur un tas de n éléments.

1.5 Structures et algorithmes supposés connus

Dans le cadre de ce cours, nous supposons connus les structures les plus basiques et leur complexité, ainsi que quelques algorithmes, que nous listons ici.

— structure abstraite des listes, des piles, des files (Figure 1.8), avec les structures concrètes par liste simplement ou doublement chainées et par tableau circulaire,

	insertion	suppression	accès
liste	en tête $O(1)$	de la tête $O(1)$	à la tête $O(1)$
pile	en tête $O(1)$	de la tête $O(1)$	à la tête $O(1)$
file	en queue $O(1)$	de la tête $O(1)$	à la tête $O(1)$

— structure abstraite des dictionnaires (Figure 1.9), structure concrète d'arbre binaire de recherche (sans équilibrage),

— structure abstraite de files de priorité, structure concrète de tas binaire,

	insertion	minimum	extraction du minimum
tas de n éléments	$O(\log n)$	$O(1)$	$O(\log n)$

— recherche dichotomique dans un tableau,

— tri d'un tableau, d'une liste, en complexité $O(n \log n)$ pour n éléments, tri en place d'un tableau,

type *file* **de** t	référence à une séquence finie d'éléments de type t
file_vide() : *pile*	$[\![\mathsf{file\_vide}()]\!] = \mathbf{ref}\langle\rangle$
est_vide(*file*) : *bool*	$[\![\mathsf{est\_vide(f)}]\!]$ = vrai si $!\,[\![f]\!] = \varnothing$, $[\![\mathsf{est\_vide(f)}]\!]$ = faux sinon
insère(t, *file*) : *void*	$[\![\mathsf{insère(elt, f)}]\!] = [\![f]\!] \leftarrow \langle e_k, e_{k-1}, \ldots, e_1, [\![\mathsf{elt}]\!]\rangle$, avec $!\,[\![\mathsf{f}]\!] = \langle e_k, \ldots, e_1\rangle$
tete(*file*) : t	$[\![\mathsf{tete(f)}]\!] = e_k$ lorsque $!\,[\![\mathsf{f}]\!] = \langle e_k, \ldots, e_0\rangle$, Précondition : $!\,[\![\mathsf{f}]\!] \neq \langle\rangle$
queue(*pile*) : *void*	$[\![\mathsf{queue(f)}]\!] = [\![\mathsf{f}]\!] \leftarrow \langle e_{k-1}, \ldots, e_1, e_0\rangle$ lorsque $!\,[\![\mathsf{f}]\!] = \langle e_k, \ldots, e_0\rangle$, Précondition : $[\![\mathsf{f}]\!] \neq \langle\rangle$

FIGURE 1.8 – La structure de données abstraite *file*

type *dictionnaire* **de** t **avec fonction** $\leq(t, t)$: *bool*	ensemble d'éléments de type t
dictionnaire_vide() : *dictionnaire*	$[\![\mathsf{dictionnaire\_vide}()]\!] = \varnothing$
est_vide(*dictionnaire*) : *bool*	$[\![\mathsf{est\_vide(dico)}]\!]$ = vrai si $[\![\mathsf{dico}]\!] = \varnothing$, $[\![\mathsf{est\_vide(dico)}]\!]$ = faux sinon
insère(t, *dictionnaire*) : *dictionnaire*	$[\![\mathsf{insère(elt, dico)}]\!] = \{\mathsf{elt}\} \cup [\![\mathsf{dico}]\!]$
appartient(t, *dictionnaire*) : *bool*	$[\![\mathsf{appartient(elt, dico)}]\!] = [\![\mathsf{elt}]\!] \in [\![\mathsf{dico}]\!]$
supprime(t, *dictionnaire*) : *dictionnaire*	$[\![\mathsf{supprime(elt, dico)}]\!] = [\![\mathsf{dico}]\!] \setminus \{\mathsf{elt}\}$,

FIGURE 1.9 – La structure de données abstraite *dictionnaire* (variante persistante)

Chapitre 2

Algorithmique des graphes

2.1 Définitions et représentations

Les graphes sont des objets mathématiques capturant la notion de relation deux à deux d'éléments. Leur importance en informatique est de premier ordre, il s'agit en effet d'un modèle utilisé pour de très nombreux problèmes, propre à l'informatique ou non. Donnons quelques exemples de questions qui sont résolues grâce à nos connaissances sur les graphes :

- Comment un appareil GPS peut-il trouver l'itinéraire le plus court ou le plus rapide entre deux points ?
- Comment *make* décide-t-il de l'ordre de compilation des fichiers source d'un projet ?
- Lors de greffes d'organes, typiquement pour les reins, le donneur et le receveur doivent être compatible. Comment affecter optimalement les reins donnés avec des demandeurs compatibles, afin de maximiser le nombre de receveur greffés ?
- Comment construire des structures en tubes d'acier dont on puisse garantir la rigidité, par exemple pour construire un échafaudage ?
- Comment une entreprise peut répartir ses ressources humaines sur différentes tâches, chacune requérant des compétences propres à certains employés seulement ?

Nous allons étudier plusieurs algorithmes, dont certains permettent de répondre à quelques unes des questions précédentes.

2.1.1 Définitions et notations

Même sans connaître la définition mathématique d'un graphe, nous avons tous déjà vu et utilisé des graphes. Pensons par exemple aux plans de transport en commun, aux diagrammes de chaines alimentaires, aux circuits électriques... Un graphe, c'est simplement des objets dont certaines paires sont reliées. Dans le cadre de ce cours, les liaisons ont une direction, on parle plus précisément de graphe orienté.

Définition 2.1.1. *Un* graphe orienté *est un couple* (V, E) *de deux ensembles (que nous supposerons toujours finis), muni de deux fonctions* $src\colon E \to V$ *et* $dst\colon E \to V$*. Les éléments de* V *sont appelés* sommets *ou parfois* nœuds*, ceux de* E *sont appelés* arcs*. Les fonctions src et dst associent à chaque arc une* source *et une* destination*, qui sont toutes deux des sommets, appelés* extremités *de l'arc.*

Les petits graphes orientés peuvent être représentés de façon picturale, en utilisant des petits disques pour les sommets, et des flèches depuis l'image de la source vers celle de la destination pour chacun des arcs. L'emplacement des disques et la forme des arcs n'a pas d'importance, on choisit en général ceux qui assurent la meilleure lisibilité. La Figure 2.1 montre un exemple de graphe orienté à 6 sommets et une possible représentation dessinée.

Le plus souvent nous choisirons $E \subset V \times V$, et src et dst seront la première et seconde projection. Ainsi, un arc noté (a, b) aura pour source a et destination b.

Arcs	src	dst
1	a	b
2	b	f
3	b	c
4	e	b
5	b	d
6	c	d
7	d	e
8	e	f

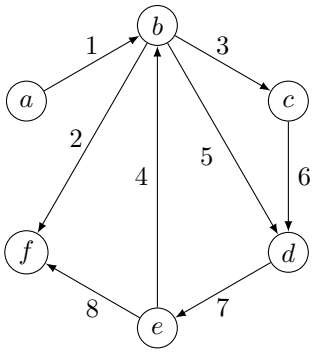


FIGURE 2.1 – Un exemple de représentation d'un graphe orienté, $V = \{a, b, c, d, e, f\}$, $E = \{1, 2, 3, 4, 5, 6, 7, 8\}$

type *graphe* **de** (*sommet*, *arc*)	Graphe
sommets(*graphe*) : **liste de** *sommet*	Liste des sommets du graphe
arcs(*graphe*) : **liste d'***arc*	Liste des arcs du graphe
tête(*arc*) : *sommet*	Tête (destination) d'un arc
queue(*arc*) : *sommet*	Queue (origine) d'un arc
arcs_sortants(*graphe*, *sommet*) : **liste d'***arc*	Liste des arcs sortant d'un sommet donné
ajoute_arc(*arc*, *graphe*) : *void*	Ajout d'un nouvel arc dans un graphe
retire_arc(*arc*, *graphe*) : *void*	Retrait d'un arc dans un graphe

FIGURE 2.2 – Interface typique d'un graphe encodé par liste d'incidence.

Définition 2.1.2. *Si un arc e a pour source ou destination un sommet u, on dit que e est* incident *à u. Si $u = src(e)$, e est un* arc sortant *de u. Si $v = dst(e)$, e est un* arc entrant *dans v.*

L'ensemble des arcs entrants dans u est noté $\delta^-(u)$. L'ensemble des arcs sortants de u est noté $\delta^+(u)$. L'ensemble des arcs incidents à u est noté $\delta(u) = \delta^+(u) \cup \delta^-(u)$. Si $U \subseteq V$ est un sous-ensemble de sommets, on note $\delta^-(U) := \{e \in E \ : \ src(u) \notin U, head(u) \in U\}$, $\delta^+(U) := \{e \in E \ : \ src(u) \in U, head(u) \notin U\}$, et $\delta(U) := \delta^+(u) \cup \delta^-(U)$.

Si $u, v \in V$ sont incidents à un même arc e, on dit que u et v sont adjacents *ou* voisins. *L'ensemble des sommets voisins d'un sommet u est noté $N(u)$.*

On notera systématiquement, sauf si cela devait créer confusion, par n le nombre de sommets du graphe dont il est question, et par m le nombre de ses arcs. La complexité de algorithmes que nous donnerons sera calculée asymptotiquement par rapport à ces deux paramètres. En général, les graphes que nous manipulerons aurons au plus un arc depuis chaque sommet vers n'importe quel autre sommet, ainsi, le nombre d'arcs m sera borné par n^2.

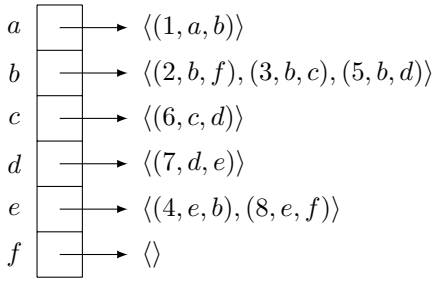
2.1.2 Représentation algorithmique par liste d'incidence

Dans le cadre de ce cours, la plupart des graphes seront décrits en donnant tous les arcs incidents à chacun des sommets. Il s'agit d'un point de vue local sur le graphe : depuis chaque sommet on peut accéder aux arcs qui y sont incidents. On peut imaginer un individu myope se balladant sur la structure même du graphe, c'est alors l'information dont il disposerait dans son exploration. Ce n'est pas cependant pas la seule façon de faire, mais au moins la plus répandue et la plus étudiée. Cela suggère l'interface minimaliste décrite en Figure 2.2, la principale fonction étant arcs_sortants, qui donne pour chaque sommet u l'ensemble $\delta^+(u)$ représenté comme une liste.

L'implémentation la plus commune de cette interface, appelé *représentation par listes d'incidence* (ou improprement *d'adjacence*), utilise un tableau indicé par les sommets du graphe, et dont l'élément d'indice i est la liste des arcs sortants pour le sommet d'indice i. Chaque arc est représenté comme une structure contenant au moins un champ pour la source et un pour la

destination.

Exemple 2.1.1. Le graphe de la Figure 2.1 donne la structure suivante :



Le graphe lui-même est encodé par une structure comprenant au moins deux champs :

— l'un de type entier, dont la valeur représente le nombre de sommets du graphe,

— l'autre est le tableau (ou une structure de dictionnaire quelconque) des listes d'incidences.

Pour utiliser un tableau, il faut pouvoir indicer les sommets par des entiers consécutifs ce qui constitue une contrainte généralement facile à satisfaire, en utilisant un type adapté pour les sommets. Comme l'accès aux arcs sortants d'un sommet sera souvent critique dans les algorithmes, il est idéal de pouvoir le faire en temps $O(1)$ dans le pire des cas.

Selon les algorithmes utilisés, nous pourrons vouloir disposer de champs supplémentaires à chaque sommet ou à chaque arc. Cette représentation n'est donc qu'une base à adapter selon les besoins précis de l'algorithme implémenté.

L'occupation mémoire d'un graphe représenté par liste d'incidence est $O(n+m)$, linéaire en le nombre d'objets (sommet ou arc) dans le graphe. L'accès aux arcs sortants est en $O(1)$, par contre tester la présence d'un arc entre deux sommets prend dans le pire des cas un temps proportionnel au nombre d'arcs sortants de la source (dans le cas où deux arcs ne peuvent avoir la même extrémité, cela peut donc prendre $O(n)$).

2.1.3 Représentation par matrice d'adjacence

On trouve souvent dans les livres des références à une autre représentation des graphes, dont nous n'aurons pas l'utilité pour ce cours mais qu'il est toujours utile de connaître.

La *représentation par matrice d'adjacence* d'un graphe consiste en un tableau à deux dimensions, chaque dimension étant indicé par l'ensemble des sommets. L'entrée de coordonnée u, v, pour deux sommets u et v, représente l'existence d'un arc de source u et de destination v. Il existe plusieurs façons d'encoder cette information, par exemple utiliser des booléens (*vrai* : cet arc existe bien), ou bien des valeurs numériques (qui peuvent représenter une donnée physique pour l'arc en question, et être fixées à 0 ou bien $-\infty$ s'il n'existe pas d'arc de u vers v).

Exemple 2.1.2. Le graphe de la Figure 2.1 serait alors représenté par la matrice (indicée de a à f dans l'ordre alphabétique)

$$\begin{pmatrix} \text{faux} & \text{vrai} & \text{faux} & \text{faux} & \text{faux} & \text{faux} \\ \text{faux} & \text{faux} & \text{vrai} & \text{vrai} & \text{faux} & \text{vrai} \\ \text{vrai} & \text{faux} & \text{faux} & \text{vrai} & \text{faux} & \text{faux} \\ \text{vrai} & \text{faux} & \text{faux} & \text{faux} & \text{vrai} & \text{faux} \\ \text{vrai} & \text{vrai} & \text{faux} & \text{faux} & \text{faux} & \text{vrai} \\ \text{vrai} & \text{faux} & \text{faux} & \text{faux} & \text{faux} & \text{faux} \end{pmatrix}$$

Avec cette représentation, l'occupation en mémoire est nécessairement en $O(n^2)$ puisque la matrice a une taille n^2. Souvent les graphes manipulés ont un nombre d'arcs sensiblement plus faible que pour le graphe complet, dans ce cas, la représentation en liste d'incidence est plus économe en mémoire puisque n'utilisant que $O(m)$ unités. C'est l'argument généralement avancé pour privilégier une représentation par rapport à l'autre. Mais il ne faut pas oublier que les opérations pouvant être facilement implémentées dans chacun de ces deux représentations ne sont pas les mêmes, et c'est ce critère qui importe le plus lors du choix d'une représentation. Ainsi :

— la représentation par liste d'incidence permet d'accéder facilement à la liste des voisins d'un sommet, mais ne permet pas de tester l'existence d'un arc entre deux sommets en temps $O(1)$,
— au contraire de la représentation par matrice d'adjacence, pour laquelle le test d'existence d'un arc est en $O(1)$, mais retrouver la liste d'incidence d'un sommet n'est pas immédiat.

Une conséquence est que la représentation par liste d'incidence est adaptée aux algorithmes reposant sur des parcours de graphe. Ces algorithmes procèdent en se déplaçant d'un sommet vers tous ses voisins. C'est le cas de tous les algorithmes que nous étudierons cette année. La représentation par matrice d'adjacence permet quand à elle d'utiliser des algorithmes basés sur l'algèbre linéaire.

2.2 Parcours de graphes, test de connectivité

Une des applications fondamentales des graphes concerne la propagation dans les réseaux. Quelques exemples : le routage de paquets dans un réseau informatique, la propagation d'une maladie dans une population, le transport de biens dans un réseau ferroviaire... Ces problèmes reposent plus particulièrement sur les possibilités pour aller d'un sommet à un autre en se déplaçant le long des arcs du graphe. Les arcs représentent donc des connexions élémentaires, et se pose alors la question de la connexion distante de plusieurs sommets. Plus formellement, l'existence de chemins.

Définition 2.2.1. *Un* chemin *dans un graphe (V, E) est une séquence d'arcs $P = e_1, \dots, e_l$ distincts tels que pour tout $i \in [1, l-1]$, $dst(e_i) = src(e_{i+1})$. La* source *de P est le sommet $u := src(e_1)$, sa* destination *est le sommet $v := dst(e_l)$ et sa* longueur *est l. On dit alors que P est un uv-chemin.*

La question la plus basique que nous puissions alors nous poser est : comment décider l'existence de chemins entre deux sommets d'un graphe orienté ? C'est l'un des objectifs des *algorithmes de parcours de graphes* que nous allons étudier dans cette partie.

Définition 2.2.2. *Un sommet v est* accessible *depuis un sommet u s'il existe un uv-chemin. Un arc $e = vw$ est* accessible *depuis un sommet u s'il existe un uw-chemin contenant e.*

Si v est un sommet accessible depuis u, on définit la distance de u vers v comme la longueur minimale d'un uv-chemin.

2.2.1 Parcours de graphes : généralités

Un *algorithme de parcours de graphe* est un algorithme qui, partant d'un sommet s d'un graphe appelé *source*, explore tous les sommets ou tous les arcs accessibles depuis cette source, et aucun autre. Pour chaque sommet u considéré hormis la source, il détermine un arc prédécesseur $\text{pred}(u)$, de telle sorte que $\text{dst}(\text{pred}(u)) = u$, et qu'il existe un entier l tel que $(\text{src} \circ \text{pred})^l(u) = s$ (la fonction *source du prédécesseur* répétée l fois donne la source). Ainsi, $(\text{src} \circ \text{pred})^l(u), (\text{src} \circ \text{pred})^{l-1}(u), \dots, \text{src}(\text{pred}(u)), u$ est un su-chemin. La fonction pred donne donc la direction (inverse) de la source depuis n'importe quel sommet accessible.

Un parcours de graphe permet donc de déterminer tous les sommets accessibles depuis la source s, et pour chacun d'eux un chemin depuis s.

La fonction pred détermine ce qu'on appelle une *arborescence* : un sous-ensemble d'arcs tel que chaque sommet possède un seul arc entrant sauf la source qui n'en a pas, et ne possédant pas de cycles.

Les algorithmes de parcours fonctionnent tous selon un principe similaire et assez naturel. L'algorithme maintient pendant son exécution deux ensembles : l'ensemble R des sommets visités (on dit aussi *parcourus*), constituant le monde connu et répertorié, et un sur-ensemble F des arcs sortant des sommets visités, que nous appelerons la frontière. Les arcs de la frontière ont donc leurs sources déjà parcourues, et si un arc a une source parcourue, mais pas sa destination, alors il doit être dans la frontière. La frontière peut aussi contenir des arcs entre deux sommets parcourus, mais ceux-ci ne sont pas utiles et pourraient être supprimés de la frontière.

```
1   fonction parcours_générique(graphe g, sommet racine) :
                tableau d'indices sommets d'(arc ou bien ⊥) :=
2     soit prédécesseur = tableau d'indices sommets(g) de (arc ou bien ⊥)
3     soit frontière := S.collection_vide()
4     soit parcouru := ref {racine}
5
6     soit fonction étends_frontière(sommet u) : void :=
7       pour tout a ∈ arcs_sortants(g, u) faire
8         S.insère(frontière, a)
9
10    soit fonction explore(arc a) : void :=
11      si tête(a) ∈ !parcouru alors retourner
12      parcouru ← !parcouru ∪ {tête(a)}
13      prédécesseur[tête(a)] ← a
14      étends_frontière(tête(a))
15
16    prédécesseur[racine] := ⊥
17    étends_frontière(racine)
18    tant que ¬ S.estVide(frontière) faire
19      explore(S.extrais(frontière))
20    retourner prédécesseur
```

FIGURE 2.3 – Algorithme générique de parcours, pour une structure d'insertion-extraction S (par exemple, une pile pour le parcours en profondeur ou une file pour le parcours en largeur).

Au début de l'exécution de l'algorithme, seule la source s est parcourue, donc tous les arcs de $\delta^+(s)$ doivent être dans la frontière. Une étape élémentaire consiste en l'exploration d'un arc de la frontière : on essaye ainsi de faire avancer les limites du monde connu en visitant un arc non-exploré. Deux cas se produisent :

- l'arc e a pour destination un sommet parcouru, dans ce cas, on a perdu un peu de temps mais on peut enlever l'arc de la frontière,
- ou bien l'arc e a une destination v qui n'est pas encore visitée. Dans ce cas, on la répertorie en ajoutant v à l'ensemble des sommets connus. Ceci nous oblige à ajouter les arcs sortants de v à la frontière, car ils peuvent aller vers des sommets inconnus (et la frontière *doit* contenir tous les arcs de source connue et de destination inconnue). On peut aussi enlever e de la frontière, puisque maintenant on connait v, et on détermine que $\text{pred}(v) = e$, puisque pour atteindre v, on est venu par e.

L'algorithme termine lorsque la frontière est vide.

L'algorithme repose sur une structure de donnée contenant la frontière, selon la structure de donnée abstraite d'*insertion-extraction*, donnée en Figure 2.4. Elle doit supporter trois opérations principales : le test du vide, l'insertion d'élément et l'extraction d'éléments (à la fois supprimer et retourner un élément). Cette structure généralise plusieurs structures abstraites connues : les listes et les piles, les dictionnaires, les files de priorités. L'algorithme de parcours est donné en Figure 2.3.

La Figure 2.5 montre un exemple de parcours d'un graphe, avec une structure d'insertion-extraction non-précisée. Le choix de l'élément extrait à chaque étape du calcul a une influence sur le résultat de l'algorithme. Les prédecesseurs de chaque sommet peuvent être différents selon ce choix. Par contre, l'ensemble des sommets sera toujours le même : c'est exactement l'ensemble des sommets accessibles. Nous le prouvons avec le lemme suivant :

Lemme 2.2.1. *L'ensemble des sommets parcourus de G depuis une source $s \in V(G)$ est l'ensemble des sommets de G accessibles depuis s.*

Démonstration. Dans un premier temps, montrons que tout sommet parcouru lors de l'exécution

type *collection* **de** t	référence à un multiensemble fini d'éléments de type t
collection_vide() : *collection*	$[\![\text{collection\_vide}()]\!] = \textbf{ref}\ \emptyset$
est_vide(*collection*) : *bool*	$[\![\text{est\_vide(c)}]\!] =$ vrai si $!\,[\![c]\!] =$ $[\![\text{est\_vide(c)}]\!] =$ faux sinon
insère(t, *collection*) : *void*	$[\![\text{insère(elt, c)}]\!] = [\![\text{c}]\!] \leftarrow! [\![\text{c}]\!] \cup \{[\![\text{elt}]\!]\}$
extrais(*collection*) : t	$\text{extrais(c)} = e \in! [\![\text{c}]\!], [\![\text{c}]\!] \leftarrow! [\![\text{c}]\!] \setminus \{e\}; e$ Précondition : $[\![\text{c}]\!] \neq \langle\rangle$

FIGURE 2.4 – La structure de données abstraite *insertion-extraction*

de l'algorithme 2.3 est bien accessible depuis la source $s = [\![\text{racine}]\!]$. Nous procédons par une induction forte sur l'itération de la boucle **tant que** de la ligne 18. Un sommet v est marqué parcouru soit en ligne 4, auquel cas il s'agit de s qui est bien accessible depuis lui-même, soit en ligne 12, auquel cas son prédécesseur est défini en ligne 13. Dans ce second cas, le prédecesseur est un arc $a = [\![\text{a}]\!]$ de destination v, par la définition de tête(a). Enfin, l'arc $a = [\![\text{a}]\!]$ provient de la frontière, comme tout arc de la frontière est inséré au moment où son origine est parcourue, l'origine u de e est parcourue avant v pendant l'exécution de l'algorithme. Par hypothèse d'induction u est accessible depuis s par un chemin P, mais alors P, e est un sv-chemin, ce qui termine l'induction.

Nous prouvons maintenant que tout sommet accessible est parcouru. Soit v un sommet accessible depuis s et $P = e_0, \ldots, e_k$ un sv-chemin. Notons $v_i = \text{dst}(e_i)$. Nous montrons par induction sur i que v_i est parcouru par l'algorithme.

Soit $i < k$, supposons donc que v_i est parcouru. Lors de l'itération pendant laquelle $v_i = [\![\text{v}]\!]$, c'est-à-dire lorsque v_i est marqué comme parcouru, e_{i+1} est un arc sortant de $[\![\text{v}]\!]$, cet arc est donc ajouté dans la frontière. Puisque l'algorithme termine lorsque la frontière est vide, il existe une itération lors de laquelle e_{i+1} est extrait. Alors les lignes 11-12 nous garantissent que v_{i+1} est parcouru au plus tard lors de cette itération. Par induction, tous les sommets du chemin P sont parcourus, dont $v = v_k$. □

Il nous faut aussi montrer que l'algorithme termine :

Lemme 2.2.2. *L'algorithme 2.3 sur un graphe G et une source s termine après au plus $2|E(G)|$ opérations sur la structure S, $|E(G)|$ insertions et $|E(G)|$ tests d'appartenance sur la structure de dictionnaire utilisée pour* parcouru *et $O(|E(G)|)$ autres opérations élémentaires.*

Démonstration. Notons d'abord que chaque sommet v est exploré au plus une seule fois (exécution des lignes 12 à 14 avec $[\![\text{v}]\!] = v$), à cause du test de la ligne 11 et de l'ajout de $[\![\text{v}]\!]$ dans parcouru en ligne 12. Du coup, chaque arc ne peut être inséré qu'au plus une fois : lors de l'appel à explore avec sa source comme argument. Le nombre d'opérations sur frontière est donc au plus $2|E(G)|$.

Chaque sommet est ajouté au plus une fois dans parcouru. De plus chaque extraction de frontière implique un test d'appartenance pour parcouru, soit au plus $|E(G)|$ en tout.

Clairement, chaque itération de la boucle **tant que** ligne 18, ou de la boucle **for** ligne 7 provoque une opération sur frontière. Donc le nombre d'autres opérations élémentaires est proportionnel aux nombres d'opérations sur frontiere. □

Corollaire 2.2.1. *Pour le choix d'une pile ou d'une file pour la structure S, et l'utilisation d'un tableau ou d'une table de dispersion pour* parcouru*, la complexité de l'algorithme de parcours est $O(|V(G)| + |E(G)|)$.*

À tout parcours de graphe, on peut associer l'ordre d'entrée dans les sommets par l'ordre dans lequel les sommets sont visités :

Définition 2.2.3. *On définit l'*ordre d'entrée *des sommets parcourus l'ordre tel que $u < v$ si u a été ajouté à l'ensemble $[\![$parcouru$]\!]$ avant v. Il s'agit donc d'un ordre total sur les sommets parcourus.*

Ainsi pour l'exemple de la Figure 2.5, l'ordre d'entrée obtenu est $3 < 1 < 2 < 5 < 4 < 7$.

Le résultat d'un algorithme de parcours est principalement le tableau des prédecesseurs. Il associe à chaque sommet visité v différent de la source, l'arc a par lequel le sommet a été découvert (de sorte que $a = [\![\mathsf{a}]\!]$ lorsque $v = [\![\mathsf{tête(a)}]\!]$ ligne 12). Une propriété immédiate de l'arc prédecesseur (u, v) de v est que u a été visité lors d'une itération précédant celle pendant laquelle v est inséré, autrement dit $u < v$ dans l'ordre d'entrée. Ainsi, la suite $\big((\mathrm{src} \circ \mathrm{pred})^k(u)\big)_k$ est une suite décroissante pour cet ordre, elle est donc finie (pred sur le dernier élément n'est pas défini), notons-la (en sens inverse) $v_0, v_1, \ldots, v_l$ avec $v_l = u$. Puisque v_0 n'a pas de prédecesseur mais fut visité (l'arc (v_0, v_1) a appartenu à la frontière), v_0 ne peut être que la source du parcours. Ainsi, $\mathrm{pred}(v_1), \mathrm{pred}(v_2), \ldots, \mathrm{pred}(v_l)$ est un sv-chemin

Définition 2.2.4. *Étant donné le parcours d'un graphe G depuis une source s, ayant produit la fonction prédecesseur pred, pour tout sommet parcouru v de G on appelle* chemin prédécesseur *le chemin $e_1, e_2, \ldots, e_l$ avec $e_i = pred(v_i)$, $v_i = src(e_{i+1})$, $v_l = v$ et $v_0 = s$. De plus, on appelle* arborescence *du parcours l'arbre enraciné en s dont les arcs sont l'image de pred.*

2.2.2 Parcours en largeur

L'algorithme de parcours 2.3 est générique, au sens où il est assez général et peut être spécialisé, par le choix de la structure d'insertion-extraction S utilisée. Ce choix dépend essentiellement des applications du parcours, et il existe une multitude de variantes. Nous en détaillons deux, les plus élémentaires qui sont aussi les plus communes.

L'algorithme de *parcours en largeur* consiste à utiliser une structure de file pour S. Un exemple de parcours en largeur est donné par la Figure 2.6. La file étant une structure FIFO (*first in, first out*), les premiers sommets explorés sont les sommets destinations d'arcs sortant de la source, autrement dit les voisins immédiats. Une fois ceux-ci explorés, l'algorithme passera aux voisins de ces voisins, et ainsi de suite. Le parcours se fait donc dans l'ordre de la distance depuis la source. Nous le formalisons ainsi :

Définition 2.2.5. *L'*algorithme de parcours en largeur *est l'algorithme de parcours obtenu en utilisant la structure de donnée abstraite des files pour l'ensemble des arcs frontières.*

Lemme 2.2.3. *Pour tout sommet parcouru u par un parcours en largeur de G depuis s, le chemin prédecesseur de u est un plus court su-chemin de G. La distance des sommets depuis la source est une fonction croissante de l'ordre d'entrée du parcours.*

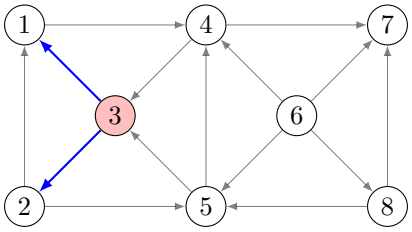
Démonstration. Notons pour tout sommet v accessible depuis s la longueur d'un plus court sv-chemin par $d_s(v)$. Rappelons que la distance de s vers v est la longueur minimale d'un sv-chemin. Nous prouvons par induction forte sur le nombre d'itérations de la boucle **tant que** ligne 18, que :

(i) si un sommet est parcouru, son chemin prédecesseur est un plus court chemin.

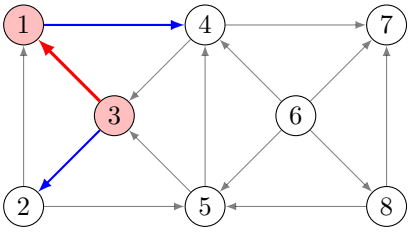
(ii) si un sommet u est parcouru, tous les sommets strictement plus proches que u de la source s ont été parcourus.

Au début de la première itération, seul s est parcouru, son prédécesseur est vide, et donc son chemin prédécesseur est le ss-chemin vide, qui est bien de longueur minimum.

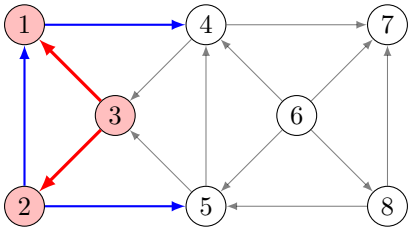
Considérons une itération quelconque et supposons l'hypothèse vraie au début de toutes les itérations précédentes et de celle-ci. Si le test de la ligne 11 est faux, il ne se passe rien et l'hypothèse reste vrai au début de l'itération suivante. Supposons donc que $[\![\mathsf{v}]\!]$ est visité pendant cette itération. Soit $u = \mathrm{src}([\![\mathsf{e}]\!])$, le chemin prédecesseur P_u de u est un plus court su-chemin. Procédons par l'absurde, supposons que P_u, e n'est pas un plus court sv-chemin, soit Q un plus court sv-chemin. Alors Q ne passe pas par u, puisqu'alors il contiendrait un su-chemin plus court que P_u. Soit e' le dernier arc de Q, et $u' = \mathrm{src}(e')$. Par l'existence de Q u' est strictement plus proche de s que u. Par hypothèse d'induction, u' était déjà parcouru lors de l'itération visitant u. Donc l'arc $u'v$ est inséré dans la frontière avant l'arc uv, ce qui contredit que v n'est pas encore parcouru lorsque uv est extrait. Ceci prouve (i).



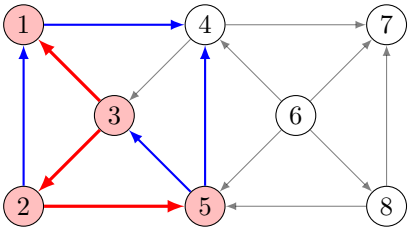
(a) Initialisation, en source 3



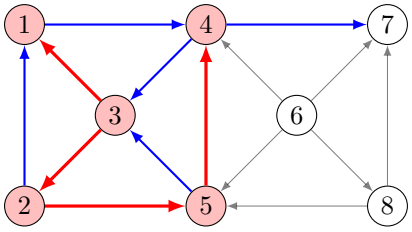
(b) avancée par (3,1) et exploration de 1



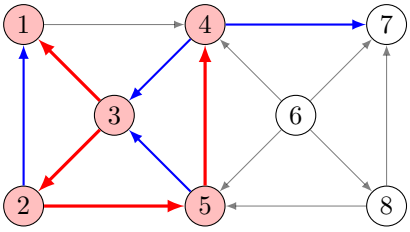
(c) avancée par (3,2) et exploration de 2



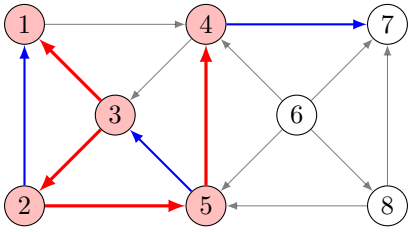
(d) avancée par (2,5), exploration de 5



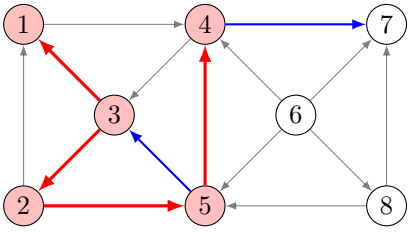
(e) avancée par (5,4) et exploration de 4



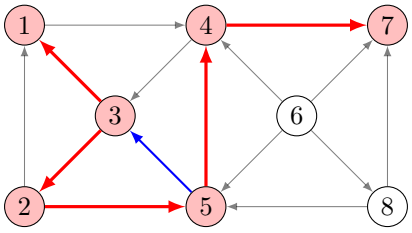
(f) avancée par (1,4), sommet 4 déjà visité



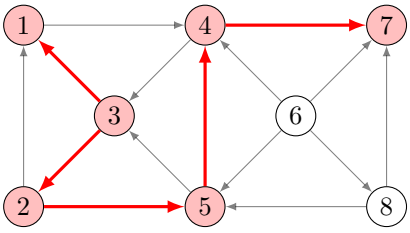
(g) avancée par (4,3), 3 déjà visité



(h) avancée par (2,1), sommet 1 déjà visité



(i) avancée par (4,7), exploration de 7



(j) avancée par (5,3), 3 déjà visité.

FIGURE 2.5 – Un exemple de parcours de graphe, avec pour source le sommet 3. En fin de parcours, les sommets 6 et 8 n'ont pas été atteint : il n'existe pas de chemin depuis 3 vers ces deux sommets.

Supposons maintenant qu'un sommet w, strictement plus loin de s que v est déjà parcouru, lors de la visite de v : $d_s(w) > d_s(v)$. Soit $w' = \text{src}(\text{pred}(w))$, alors $d_s(w') = d_s(w) - 1 > d_s(v) - 1 = d_s(u)$. Donc w' n'était pas parcouru lors de l'itération visitant u par hypothèse d'induction. L'arc (w', w) a donc été ajouté à la frontière après l'arc (u, v), contradiction. Ceci prouve (ii) et termine l'induction. □

L'algorithme de parcours en largeur permet donc de calculer les plus courts chemins depuis une source. Plus exactement, il s'agit de plus courts chemins, où la longueur d'un chemin est donné par son nombre d'arcs. Nous verrons comment généraliser ce résultat lorsque chaque arc possède une longueur et que la longueur d'un chemin est la somme des longueurs de chaque arc. Nous utiliserons aussi les parcours en largeur pour concevoir un algorithme de flot.

2.2.3 Parcours en profondeur

Une autre variation simple de l'algorithme de parcours consiste à utiliser une pile, donc une structure LIFO (*last in, first out*). Ceci change radicalement l'ordre de traitement des sommets, puisqu'il est tout à fait possible qu'un sommet adjacent à la source soit parcouru en dernier par exemple. Il s'agit plutôt d'un algorithme de type *backtracking*, où l'on avance tant qu'il existe des sommets non-visités, sinon on repart vers l'arrière.

Définition 2.2.6. *L'*algorithme de parcours en profondeur *est l'algorithme de parcours obtenu en utilisant la structure de donnée abstraite des piles pour l'ensemble des arcs frontières.*

Cet algorithme possède une propriété intéressante : il peut être utiliser par un humain dans un labyrinthe, contrairement au parcours en largeur. En effet, lors d'un parcours en largeur, deux arcs très éloignés l'un de l'autre peuvent être extraits consécutivement, tant qu'ils sont tous les deux à une même distance de la source. L'algorithme *saute* donc d'un endroit à l'autre du graphe. Au contraire, dans le parcours en profondeur, on progresse tant que possible en suivant des arcs successifs, formant un chemin. Si ce n'est plus possible, on revient en arrière sur ce chemin jusqu'à trouver un nouvel embranchement possible. L'algorithme reste donc toujours au niveau local. Un exemple d'exécution de l'algorithme de parcours en profondeur est donné en Figure 2.7.

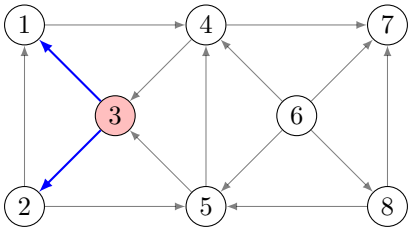
Une autre particularité du parcours en profondeur est de pouvoir être codé sans mentionner explicitement la structure d'insertion-extraction. En effet, il nous faut utiliser une pile, mais nous pouvons nous servir de la pile d'appel récursif, ce que nous faisons avec l'algorithme de la Figure 2.8. Dans cet algorithme, c'est la fonction explore qui est récursive. Elle prend en argument un arc, et essayer de visiter la tête de cet arc.

Comme dans l'algorithme de parcours générique, le parcours en profondeur définit un ordre d'entrée dans les sommets parcourus. Contrairement à l'algorithme générique, dans lequel on ajoute tous les sommets à la frontière avant de passer au sommet suivant, le traitement de la visite d'un sommet n'est pas groupé temporellement dans la version récursive du parcours en profondeur : on ajoute un arc sortant, on le visite récursivement, puis on ajoute le deuxième arc sortant, on le visite récursivement, et ainsi de suite jusqu'au dernier arc sortant. Il devient alors intéressant de définir l'ordre de sortie des sommets :

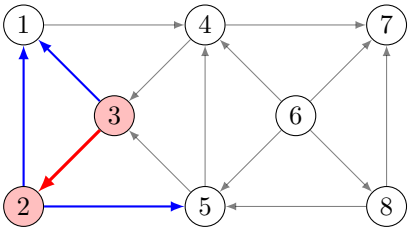
Définition 2.2.7. *L'ordre de sortie des sommets d'un parcours en profondeur est l'ordre total sur les sommets parcourus tel que $u < v$ si $s = v$ ou bien la fin de l'exécution de la ligne 9 lors de l'appel de* explore *lors duquel $[\![v]\!] = u$ est parcouru précède la fin de l'exécution de la ligne 9 lors de l'appel de* explore *lors duquel v est parcouru.*

Lemme 2.2.4. *À tout moment de l'exécution de l'algorithme de parcours en profondeur, Figure 2.8, la pile d'appels des arguments de la fonction* explore *est un sv-chemin, avec $s = [\![s]\!]$ et $v = [\![v]\!]$.*

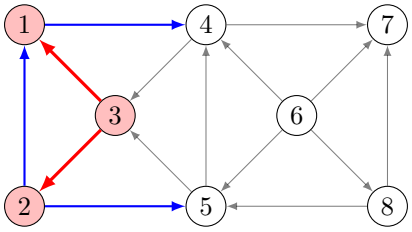
Démonstration. Lors d'un appel en ligne 10, la pile est initialement vide et on lui ajoute un arc sortant de s, donc la pile décrit bien un chemin de source s et de destination la tête de l'arc empilé. Par induction sur le nombre d'appels récursifs, on analyse le cas d'un appel récursif en ligne 9.



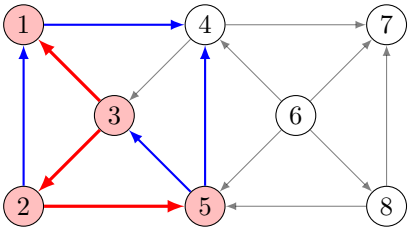
(a) Frontière : $\langle (3,1),(3,2)\rangle$



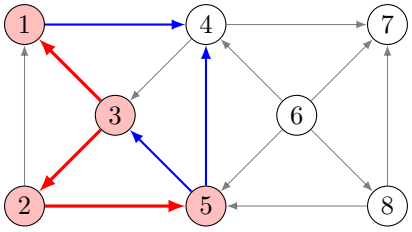
(b) Frontière : $\langle (2,1),(2,5),(3,1)\rangle$



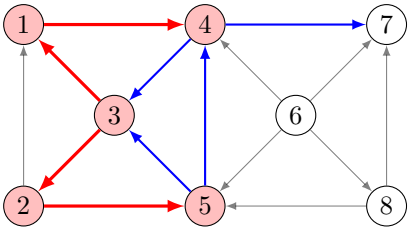
(c) Frontière : $\langle (1,4),(2,1),(2,5)\rangle$



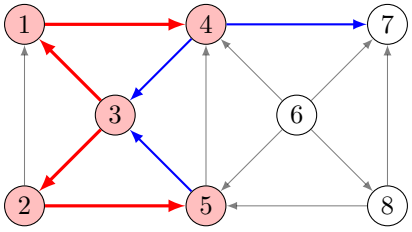
(d) Frontière : $\langle (5,3),(5,4),(1,4),(2,1)\rangle$



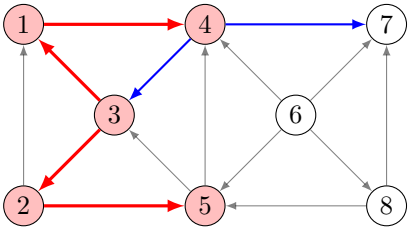
(e) Frontière : $\langle (5,3),(5,4),(1,4)\rangle$



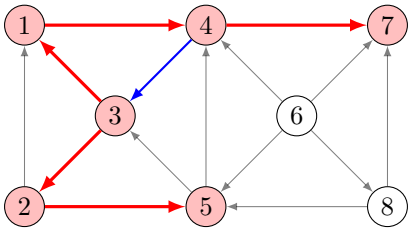
(f) Frontière : $\langle (4,3),(4,7),(5,3),(5,4)\rangle$



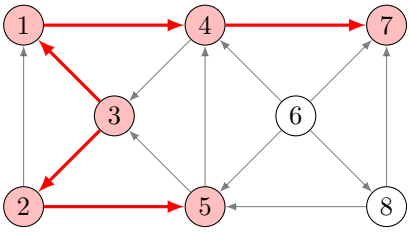
(g) Frontière : $\langle (4,3),(4,7),(5,3)\rangle$



(h) Frontière : $\langle (4,3),(4,7)\rangle$

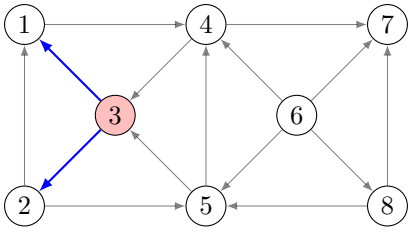


(i) Frontière : $\langle (4,3)\rangle$

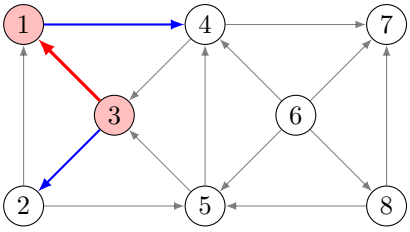


(j) Frontière : $\langle\rangle$

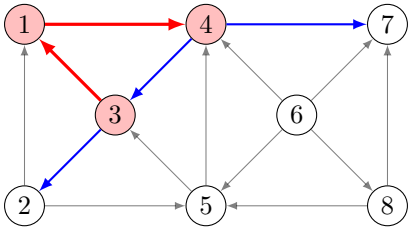
FIGURE 2.6 – Un parcours du même graphe en largeur d'abord.



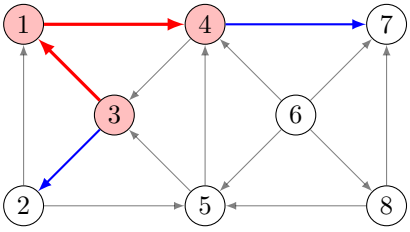
(a) Frontière : $\langle (3,1),(3,2) \rangle$



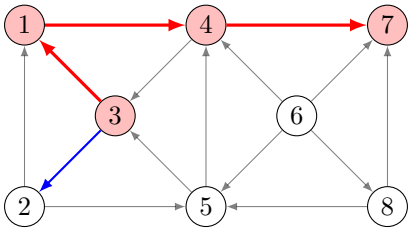
(b) Frontière : $\langle (1,4),(3,2) \rangle$



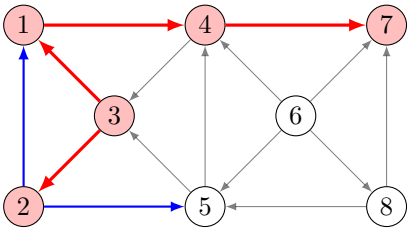
(c) Frontière : $\langle (4,3),(4,7),(3,2) \rangle$



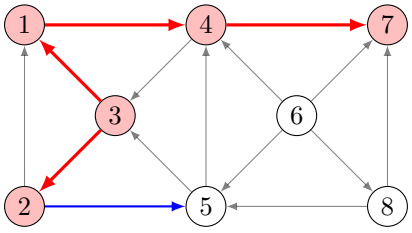
(d) Frontière : $\langle (4,7),(3,2) \rangle$



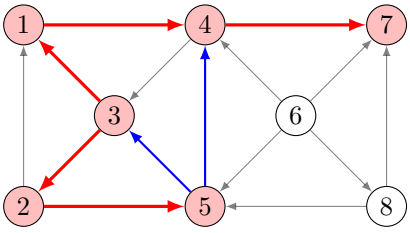
(e) Frontière : $\langle (3,2) \rangle$



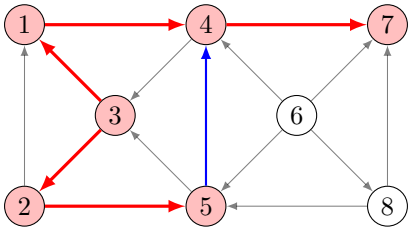
(f) Frontière : $\langle (2,1),(2,5) \rangle$



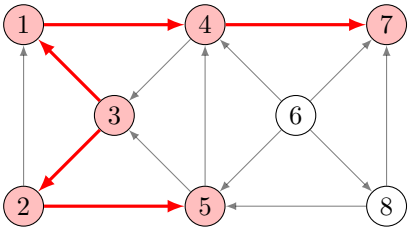
(g) Frontière : $\langle (2,5) \rangle$



(h) Frontière : $\langle (5,3),(5,4) \rangle$



(i) Frontière : $\langle (5,4) \rangle$



(j) Frontière : $\langle \rangle$

FIGURE 2.7 – Un parcours du même graphe en profondeur d'abord. Nous obtenons dans ce cas précis la même arborescence pour pour le parcours en largeur, mais ce n'est pas toujours le cas.

```
1   fonction parcours_en_profondeur(graphe g, sommet s) : tableau d'indice sommet d'(arc ou bien ⊥)
2     soit predecesseur := tableau d'indice sommets(g) de (arc ou bien ⊥)
3     soit parcouru := {s}
4     predecesseur[s] ← ⊥
5     soit fonction explore(arc e) : void =
6       soit v := tete(e)
7       si v ∉ parcouru alors
8         parcouru ← parcouru ∪ {v}; predecesseur[v] ← e
9         pour tout arc ∈ arc_sortants(g, v) faire explore(arc)
10    pour tout arc ∈ arc_sortants(g, s) faire explore(arc)
11    retourner predecesseur
```

FIGURE 2.8 – Version récursive du parcours en profondeur : cette version utilise la pile d'appels récursifs comme structure d'insertion-extraction.

Cet appel ajoute un arc $e = [\![\mathsf{e}]\!]$ avec $\mathrm{src}(e) = [\![\mathsf{v}]\!]$. On obtient donc bien que la pile décrit un su-chemin avec $u = \mathrm{dst}([\![\mathsf{e}]\!])$. □

Ceci justifie la définition suivante :

Définition 2.2.8. *À tout moment de l'exécution du parcours en profondeur, on appelle* chemin d'appel *le chemin décrit par la pile d'appel récursif.*

Pour distinguer les deux ordres, on note $<_e$ l'ordre d'entrée et $<_s$ l'ordre de sortie.

Lemme 2.2.5. *Soit $v \in V(G)$, l'ensemble $\{u \in V(G) : v <_e u \wedge u <_s v\}$ est exactement l'ensemble des sommets accessibles depuis v dans le graphe restreint aux sommets non-parcourus au moment où v est découvert (ligne 7 algorithme de la Figure 2.8 avec $[\![\mathsf{v}]\!] = v$).*

Démonstration. Simplement, à ce moment-là on se trouve dans les conditions initiales du premier appel de la fonction explore pour le graphe restreint en question, avec la différence qu'il existe éventuellement des arcs vers des sommets parcourus, mais ceux-ci sont ignorés. Par la correction des algorithmes de parcours, le lemme s'ensuit. □

2.2.4 Parcours itérés

Faire un parcours d'un graphe depuis une source s permet de parcourir tous les sommets accessibles depuis s, mais les autres sommets sont ignorés. Plusieurs algorithmes ont néanmoins besoin de parcourir tous les sommets. Dans ce cas, la solution est simple : si un sommet n'est pas visité par le premier parcours, on recommence un second parcours depuis ce sommet.

Bien sûr le deuxième parcours ne doit pas redécouvrir les sommets parcourus lors du premier parcours. Les différents parcours successifs qui sont nécessaires utilisent une même structure de donnée pour l'ensemble parcouru et pour le tableau prédecesseur.

L'algorithme de parcours itéré est donc donné par la Figure 2.9.

La complexité de cet algorithme est $\Theta(|V(G)| + |E(G)|)$ puisque chaque sommet est parcouru exactement une fois et chaque arc considéré exactement deux fois. Nous pouvons aussi décrire une version itérée récursive du parcours en profondeur de la même façon, Figure 2.10.

2.3 Tri topologique

Nous utilisons un parcours en profondeur pour résoudre le problème de trouver dans quel ordre `make` doit compiler les fichiers d'un projet. Pour cela, on peut écrire le graphe des fichiers : l'ensemble des sommets est l'ensemble des fichiers, et il existe un arc du fichier `a` vers le fichier `b` si `b` doit être compilé après `a`. Ce graphe est appelé graphe des dépendances.

Puisque nous cherchons à ordonner les fichiers dans l'ordre dans lequel nous souhaitons les compiler, nous voulons un ordre total $<$ tel que pour tout arc (u, v), $u < v$.

```
1   fonction parcours_générique(graphe g) : tableau d'indices sommets d'(arc ou bien ⊥) :=
2     soit prédécesseur = tableau d'indices sommets(g) de (arc ou bien ⊥)
3     soit frontière := S.vide()
4     soit parcouru := ref {}
5
6     soit fonction étends_frontière(sommet u) : void :=
7       pour tout a ∈ arcs_sortants(g, u) faire
8         S.insère(frontière, a)
9
10    soit fonction explore(arc a) : void :=
11      si tête(a) ∈ !parcouru alors retourner
12      parcouru ← !parcouru ∪ {tête(a)}
13      prédécesseur[tête(a)] ← a
14      étends_frontière(tête(a))
15
16    soit fonction repars_depuis(sommet racine) : void :=
17      prédécesseur[racine] := ⊥
18      parcouru ←!parcouru ∪ {racine}
19      étends_frontière(racine)
20      tant que ¬ S.estVide(frontière) faire
21        explore(S.extrais(frontière))
22
23    pour tout sommet ∈ sommets(g) faire
24      si ¬sommet ∈!parcouru alors repars_depuis(sommet)
25    retourner prédécesseur
```

FIGURE 2.9 – Parcours générique itéré. On retrouve les mêmes fonctions que le parcours générique simple, la différence est que le parcours simple fait un seul appel à la fonction repars_depuis avec la racine voulue en argument. 2.3.

```
1   fonction parcours_itéré_prof(graphe g, sommet s) : tableau d'indice sommet d'(arc ou bien ⊥)
2     soit predecesseur := tableau d'indice sommets(g) de (arc ou bien ⊥)
3     soit parcouru := {s}
4     predecesseur[s] ← ⊥
5     soit fonction explore(arc e) : void =
6       soit v := tete(e)
7       si v ∉ parcouru alors
8         parcouru ← parcouru ∪ {v}; predecesseur[v] ← e
9         pour tout arc ∈ arc_sortants(g, v) faire explore(arc)
10    pour tout s ∈ sommets(g) faire
11      si s ∉ parcouru alors
12        parcouru ← parcouru ∪ {s}
13        pour tout arc ∈ arc_sortants(g, s) faire explore(arc)
14    retourner predecesseur
```

FIGURE 2.10 – Version itérée récursive du parcours en profondeur.

Définition 2.3.1. *Pour un graphe orienté G, un* ordre topologique *est un ordre total $<$ sur $V(G)$ tel que pour tout arc (u, v), $u < v$.*

Nous commençons par chercher quand un tel ordre existe.

Définition 2.3.2. *Un* cycle *dans un graphe orienté est un chemin de longueur strictement positive dont les deux extrémités sont identiques. Un graphe orienté G est* acyclique *s'il ne possède pas de cycle.*

Lemme 2.3.1. *G possède un ordre topologique si et seulement si G est acyclique.*

Démonstration. Supposons que G possède un ordre topologique $<$ et prouvons que G est acyclique. Par l'absurde, supposons que G possède un cycle $e_1, \ldots, e_l$, et posons $e_i = (v_{i-1}, v_i)$. Alors $v_0 < v_1 < \ldots < v_l = v_0$, donc $v_0 < v_0$, ce qui contredit le fait que $<$ est un ordre.

Supposons maintenant que G est acyclique. Alors il existe un sommet u qui n'a pas d'arc sortant : par l'absurde, si ce n'est pas le cas, on construit une suite infinie d'arcs consécutifs $(u_0, u_1), (u_1, u_2), \ldots$. Puisque le nombre de sommets est fini, il existe un sommet v qui apparait au moins deux fois $v = u_i = u_j$. En prenant $i < j$ et $j-i$ minimum $(u_i, u_{i+1}), (u_{i+1}, u_{i+2}), \ldots, (u_{j-1}, u_j)$ est un cycle, contradiction. Donc u existe. Alors le graphe G' obtenu en supprimant u et $\delta(u)$ de G est acyclique. Par induction sur le nombre de sommets, G' possède un ordre topologique. En étendant cet ordre à u tel que $u > v$ pour tout $v \in V(G) \setminus \{u\}$, nous obtenons un ordre topologique. □

Ce lemme explique pourquoi une compilation peut échouer avec un message d'erreur contenant les termes "dépendance cyclique". Ce message indique la présence d'un cycle dans le graphe de dépendance.

Nous sommes donc ramené aux questions :

— décider s'il existe un cycle dans un graphe orienté,

— trouver un ordre topologique dans un graphe acyclique.

Les deux réponses se basent sur le parcours en profondeur.

Théorème 2.3.1. *G possède un cycle si et seulement si à un moment de l'exécution de l'algorithme de parcours itéré en profondeur, le chemin d'appel possède un cycle.*

Démonstration. Si le chemin d'appel contient un cycle, certainement G contient un cycle.

Si G possède un cycle $(u_0, u_1), (u_1, u_2), \ldots, (u_{l-1}, u_l)$, avec $u_0 = u_l$. Sans perte de généralité disons que u_0 est le premier sommet parcouru parmi les sommets de ce cycle. Par le Lemme 2.2.5, u_{l-1} est découvert alors que u_0 est sur le chemin d'appel. Donc $\mathsf{explore}(\mathsf{arc})$, pour $[\![\mathsf{arc}]\!] = (u_{l-1}, u_l)$, est appelé à ce moment. Alors le chemin d'appel passe deux fois par le sommet $u_0 = u_l$ et donc contient un cycle. □

En gardant un tableau pour noter les sommets dans le chemin d'appel, on peut tester l'existence d'un cycle en temps $\Theta(|V(G)| + |E(G)|)$.

Théorème 2.3.2. *Si G est un graphe acyclique, l'ordre de sortie $<_s$ d'un parcours itéré en profondeur est l'inverse d'un ordre topologique.*

Démonstration. Soit $uv \in E(G)$. On doit montrer que $v <_s u$.

Premier cas, si $u <_e v$, par le Lemme 2.2.5, $v <_s u$ comme il se doit.

Dans l'autre cas, $v <_e u$. Par contradiction, supposons que $u <_s v$. De nouveau par le Lemme 2.2.5, u est accessible depuis v, il existe un vu-chemin et un arc uv donc il existe un cycle, contradiction. Donc $v <_s u$ dans ce cas aussi. □

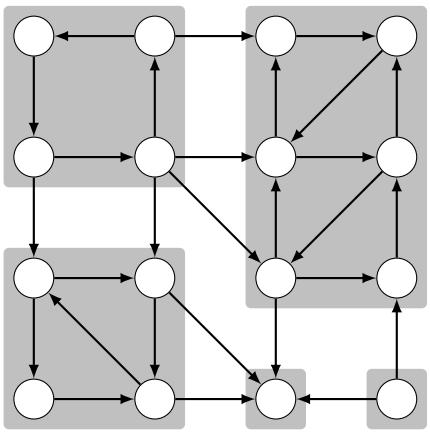
Là encore, le parcours en profondeur modifié pour attribuer un ordre de sortie (ou bien en numérotant les sommets, ou bien en retournant une liste de sommets ordonnées comme cela sera utile pour la section suivante) garde sa complexité de $\Theta(|V(G)| + |E(G)|)$.

2.4 Recherche de composantes fortement connexes

Nous appelons *composante fortement connexe* de G tout ensemble maximal de sommets C telle que pour toute paire $(u, v) \in C$, u est accessible depuis v et v est accessible depuis u. Nous avons donc que pour tout sommet $w \notin C$, ou bien w n'est pas accessible depuis les sommets de C, ou bien aucun sommet de C n'est accessible depuis w (possiblement les deux).

Définition 2.4.1. *$u \in V(G)$ et $v \in V(G)$ sont* mutuellement accessibles *dans G si u est accessible depuis v et v est accessible depuis u dans G.*

Les composantes fortement connexes sont donc les classes d'équivalence de la relation *être mutuellement accessible*. L'exemple suivant montre les cinq composantes fortement connexes d'un graphe :



Pour trouver les composantes fortement connexes, plusieurs algorithmes existent qui sont basés sur des parcours en profondeur, notamment celui de Tarjan et celui de Kosaraju que nous étudions maintenant.

— Nous procédons à un premier parcours itéré en profondeur de G, ceci nous donne un ordre de sortie sur les sommets. Il s'agit donc du même algorithme que celui permettant de trouver un tri topologique, mais que nous utilisons sur un graphe qui n'est pas nécessairement acyclique.

— Dans cet ordre en sens décroissant (donc en partant du dernier sommet trouvé par la première étape), nous exécutons un deuxième parcours itéré en profondeur, mais dans $\overleftarrow{G}$. Il s'agit du graphe obtenu en inversant la direction de chaque arc dans G. Chaque parcours élémentaire produit une liste de sommets qui est une composante fortement connexe de G.

La Figure 2.11 montre un exemple d'exécution de l'algorithme de Kosaraju.

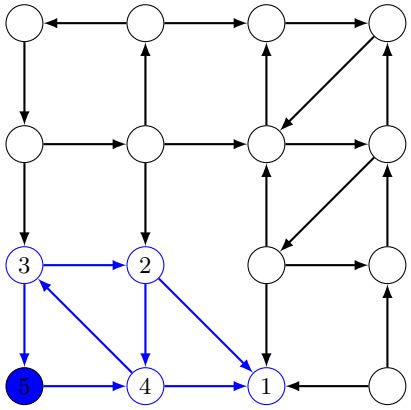
Une clé pour comprendre l'algorithme est le lemme suivant :

Lemme 2.4.1. *Les graphes G et $\overleftarrow{G}$ ont les mêmes composantes fortement connexes.*

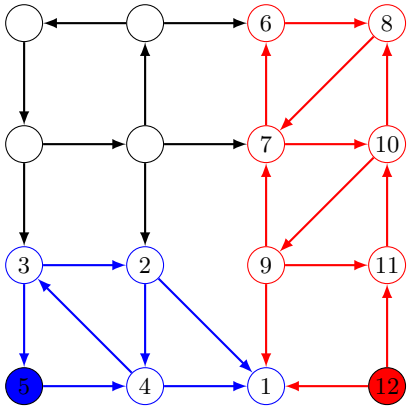
Démonstration. Un uv-chemin dans G donne lieu à un vu-chemin dans $\overleftarrow{G}$, et inversement. Donc deux sommets sont mutuellement accessibles dans G ssi ils le sont dans $\overleftarrow{G}$. □

La première étape de l'algorithme permet de construire un ordre topologique sur les composantes fortement connexes. Il suffirait ensuite de parcourir chacune de ses composantes en ordre croissant. Ainsi chaque composante serait découverte séparément. Malheureusement le premier ordre trouvé est d'une certaine façon mal orienté. Donc le second parcours se fait dans $\overleftarrow{G}$ qui heureusement possède les mêmes composantes connexes, et pour lequel l'ordre décroissant convient.

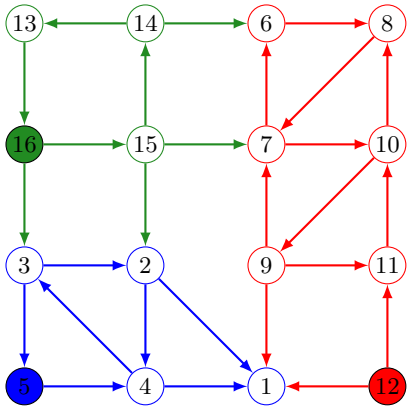
Lemme 2.4.2. *Dans G, si v est accessible depuis u, lors d'un parcours itéré, v est découvert au plus tard lors de l'itération découvrant u.*



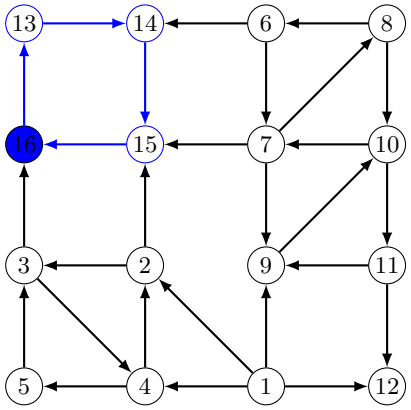
Parcours en profondeur du sommet bleu



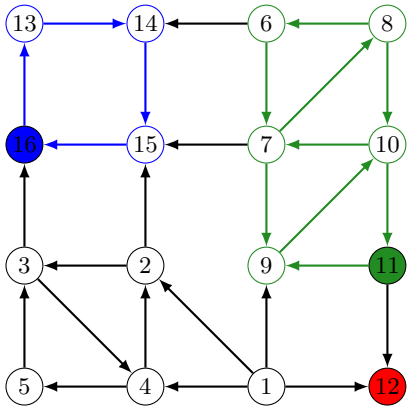
On repart du sommet rouge



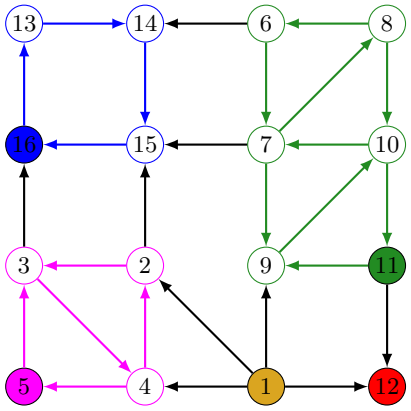
On repart du vert, ceci termine la phase 1



Première composante fortement connexe



Deuxième (rouge) et troisième (verte)



Quatrième (fuschia) et Cinquième (jaune)

FIGURE 2.11 – Exemple d'exécution de l'algorithme

Démonstration. En effet, si le sommet u est découvert lors d'une itération partant d'une source s, alors u est accessible depuis s. De plus, v est accessible depuis u : il existe donc un chemin de sommets $u, u_1, u_2, \ldots, u_k, v$.

Premier cas : s'il existe $i \in [\![1, k]\!]$ tel que u_i était parcouru avant l'itération de s, alors v était déjà découvert.

Deuxième cas : sinon, lors de l'itération partant de s, ce chemin montre que v est accessible depuis u et donc s par des sommets non-découverts au début de l'itération partant de s. Par correction du parcours en profondeur, v est donc découvert lors de l'itération partant de s. □

Lemme 2.4.3. *Lors d'un parcours itéré, l'ensemble des sommets découverts par chaque itération est l'union de une ou plusieurs composantes fortement connexes.*

Démonstration. On applique deux fois le Lemme 2.4.2 : si u et v sont mutuellement accessibles, ils sont découverts lors de la même itération. □

Lemme 2.4.4. *Soit $<_s$ l'ordre de sortie d'un parcours itéré en profondeur de G. Soit C_1 et C_2 deux composantes connexes distinctes, $u_1 \in C_1$ et $u_2 \in C_2$, et u_2 est accessible depuis u_1. Alors il existe $v \in C_1$ tel que pour tout $u \in C_2$, $u <_s v$.*

Démonstration. Soit s le premier sommet découvert de $C_1 \cup C_2$. On distingue les deux cas.

Si $s \in C_1$, alors tous les sommets de $C_1 \cup C_2$ sont accessibles depuis s, par le Lemme 2.2.5, pour tout $u \in C_1 \cup C_2$, $u \leq_s s$.

Si $s \in C_2$, par le Lemme 2.2.5, pour tout $u \in C_2 \setminus \{s\}$ et pour tout $v \in C_1$, $u <_s s$ (car u est accessible depuis s, ils sont dans la même composante connexe), et $s <_s v$ (car v n'est pas accessible depuis s sinon C_1 et C_2 formeraient une seule composante fortement connexe, et s n'est pas dans le chemin d'appel lorsque v est découvert donc on n'a pas $v \leq_s s$, d'où $s <_s v$ puisque $<_s$ est un ordre total). On conclut par transitivité de l'ordre $<_s$. □

Définissons l'ordre $<_C$ sur les composantes connexes de G par $C_1 <_C C_2$ si $\max_{<_s} C_1 < \max_{<_s} C_2$, et le graphe G_C dont les sommets sont les composantes connexes de G et (C_1, C_2) est un arc de G_C s'il existe $u \in C_1$, $v \in C_2$ et $(u, v) \in E(G)$. Alors le lemme précédent montre que $<_C$ est un ordre topologique de G_C.

Théorème 2.4.1. *L'algorithme de Kosaraju est correct : il calcule correctement les composantes fortement connexes d'un graphe orienté.*

Démonstration. Par induction sur les itérations de la deuxième phase de l'algorithme, on montre que les ensembles $C_1, C_2, \ldots, C_k$ trouvés jusque-là sont des composantes fortement connexes de $\overleftarrow{G}$.

Pour $k \geq 0$, supposons l'hypothèse d'induction vraie, et démontrons que C_{k+1} est une composante fortement connexe.

Par le Lemme 2.4.3, C_{k+1} est l'union de plusieurs composantes fortement connexes, notons les $D_1 >_C \ldots >_C D_i$. Par le choix de l'algorithme et par le Lemme 2.4.4, la source choisie pour cette itération est dans D_1 la plus grande composante pour l'ordre $<_C$. Donc il n'existe pas d'arcs dans $\overleftarrow{G}$ de D_1 vers D_j pour $j \geq 2$, donc $i = 1$. Ce qui prouve que C_{k+1} est une composante fortement connexe de G. □

2.5 Flots maximums

Nous venons d'étudier les parcours de graphes et plusieurs algorithmes qui se réduisent à un parcours de graphe dans un ordre précis, et pendant lesquels des calculs élémentaires sont produits à chaque étape du parcours. Nous allons maintenant aborder un problème plus complexe, qui utilise toujours un algorithme de parcours, mais dont le parcours est l'étape élémentaire, qui sera répété autant que nécessaire.

Les flots dans les graphes modélisent le déplacement d'une ressource dans un réseau. Plus que cela, les flots permettent de résoudre de nombreux problèmes pour lesquels une ressource est

présente en quantité fixe et doit être affecté à diverses tâches. Par le résultat de dualité entre flots et coupes que nous aborderons, ils sont aussi la base des problèmes de *clustering*, qui concernent le découpage d'un ensemble en diverses parties regroupant des éléments similaires.

2.5.1 Définition

Une façon intuituive de se représenter les flots est d'imaginer un réseau de tuyauterie, dans lequel on souhaite faire passe un maximum de liquide entre deux points, chaque tuyau ayant un débit maximum fixe. La tuyauterie, c'est le graphe orienté, la capacité d'un arc représente le débit par un réel positif, et le liquide est le flot décrivant quel est le débit utilisé pour chaque arc :

Définition 2.5.1. *Soit G un graphe orienté, s et t deux sommets de G, et $c\colon E(G) \to \mathbb{R}^+$ une fonction de capacité. Un* st-flot *de G, c est une fonction $f\colon E(G) \to \mathbb{R}^+$ qui vérifie les deux propriétés suivantes :*

Loi de capacité *pour tout arc $e \in E(G)$, $f(e) \leq c(e)$,*

Loi de conservation *pour tout sommet $v \in V(G) \setminus \{s, t\}$,*

$$\sum_{e \in \delta^+(v)} f(e) = \sum_{e \in \delta^-(v)} f(e)$$

La quantité $\sum_{e\in\delta^-(v)} f(e) - \sum_{e\in\delta^+(v)} f(e)$ pour un sommet v est appelé excès *du sommet v et est notée excès(v). La* valeur *d'un st-flot f est l'excès en t, excès(t). Les sommets s et t sont appelés la* source *et le* puits *du flot.*

La loi de conservation peut se réécrire excès$(v) = 0$ si $v \notin \{s, t\}$. Cette loi impose que toute quantité de flot rentrant dans un sommet doit en sortir, et pas plus : rien ne se perd, rien ne se crée, tout se transporte ! Seule la source émet une quantité de flot, et le puits en absorbe.

Définition 2.5.2. *Le problème du* flot maximum, *étant donné G, $c : E(G) \to \mathbb{R}^+$ et s et t deux sommets de G, consiste en trouver un st-flot dans G de valeur maximum.*

Autrement dit, on veut transférer une quantité maximum de flot depuis la source s vers le puits t. Notamment, l'excès de flot en t est la négation de l'excès en s (autrement dit égal au flot généré par s) :

Lemme 2.5.1. *Pour un st-flot f de G, c, excès$(s) = -$excès(t).*

Démonstration.

$$\begin{aligned}
\text{excès}(t) + \text{excès}(s) &= \sum_{v \in V(G)} \text{excès}(v) \\
&= \sum_{v \in V(G)} \left(\sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e) \right) \\
&= \sum_{e \in E(G)} f(e) - f(e) = 0
\end{aligned}$$

où la première égalité consiste à ajouter l'excès de chaque autre sommet, tous nuls, la seconde inégalité développe la définition de l'excès, la troisième inverse l'ordre des sommations. En effet, chaque arc est une fois arc sortant d'un sommet, et un fois arc entrant d'un sommet. □

2.5.2 L'algorithme de Ford-Fulkerson

Nous cherchons maintenant à calculer le flot maximum d'un graphe G avec une fonction de capacité c. Une proposition naïve est de partir du flot qui attribue 0 à chaque arc, et de l'améliorer petit à petit :

Proposition 2.5.1. *La fonction $f : e \mapsto 0$ qui associe 0 à chaque arc de G est un flot. On l'appelle le* flot nul.

Démonstration. Pour tout arc $e \in E(G)$, $0 = f(e) \leq c(e)$ (loi de capacité). Pour tout sommet u, excès$(u) = 0$ (loi de conservation). □

Pour améliorer le flot, il faut augmenter la valeur de f sur certains arcs. Nous sommes maintenant guidés par les deux lois : la loi de capacité nous empêche d'augmenter trop la valeur des arcs, c'est la plus facile à prendre en compte. La loi de conservation nous dit que si on augmente le flot sur l'arc uv, alors (sauf si $u = s$ ou $v = t$), il faut aussi trouver un arc (au moins) de destination u et un arc (au moins) de source v à augmenter d'autant. Nous constatons donc que chaque augmentation est un problème global : nous ne pouvons pas augmenter juste le flot d'un seul arc, mais il nous faut un ensemble d'arcs.

Une première tentative, insuffisante. Nous voyons que pour tout sommet (hors s et t), si on augmente la quantité de flot sortant, il faut aussi augmenter la quantité de flot entrant. Et inversement ! Le plus simple est donc de choisir un arc entrant et un arc sortant, et d'augmenter le flot d'autant sur chaque arc. Autrement dit, les arcs que nous choisissons doivent formé un st-chemin !

Nous pouvons donc esquisser notre premier algorithme : à partir d'un flot, trouver un st-chemin et augmenter le flot sur ce st-chemin. La quantité de flot qui peut être ajoutée au chemin $P = e_1, e_2, \ldots, e_l$ sans violer la loi de capacité est donc $\Delta(P) := \min_{i \in \{1,\ldots,l\}} c(e_i) - f(e_i)$. Puisque nous voulons vraiment augmenter la valeur du flot, nous voulons $\Delta(P)$ strictement positif, ce qui impose de trouver un st-chemin dans le graphe restreint aux arcs e tels que $c(e) < f(e)$. Nous pouvons ensuite répéter autant de fois que nécessaire cette opération, jusqu'à ce qu'il n'existe plus de tel st-chemin.

Définition 2.5.3. *Pour un graphe G, une fonction de capacité $c : E(G) \to \mathbb{R}^+$ et un flot f de G, c, on dit que l'arc e est* saturé *si $f(e) = c(e)$. La* capacité résiduelle *d'un arc e de G est $c(e) - f(e)$. Un arc est donc saturé ssi sa capacité résiduelle est* 0.

Nous devons donc trouver des st-chemins dans le graphe restreint aux arcs non saturés, et augmenter le flot sur les arcs de ce chemin par le minimum de la capacité résiduelle des arcs du chemin (afin de ne pas violer la contrainte de capacité). La Figure 2.12 montre un exemple d'exécution de cet algorithme sur un graphe à 6 sommets. Lors de la première étape, l'algorithme choisit le chemin du haut et augmente le flot de 1 puisque c'est la capacité de l'arc intermédiaire. Ensuite l'algorithme utilise le chemin du bas, chaque arc ayant une capacité résiduelle de 1. Cela donne le flot en bas à gauche de la figure. Suite à cette augmentation, il n'existe plus de st-chemin composé uniquement d'arcs non-saturés, ce qui termine l'algorithme.

Malheureusement, ce flot n'est pas maximum : il a une valeur de 2, mais sur le même graphe, il existe un flot d'une valeur de 3, comme le montre la Figure 2.13. Notre premier algorithme n'est donc pas correct : il ne calcule pas le flot maximum.

Vers un algorithme correct. Pour bien comprendre pourquoi notre premier algorithme échoue, il faut analyser le graphe des arcs non-saturés, en bas à droite de la Figure 2.12. L'ensemble des sommets accessibles depuis s représente les sommets vers lesquels il est toujours possible d'envoyer du flot. D'une certaine façon, nous pouvons considérer que ces sommets ont donc du flot potentiellement disponible à envoyer : ils abondent de flot. À l'inverse, ceux depuis lesquels t est accessibles sont les sommets qui pourraient envoyer du flot vers t, s'ils en disposaient. Cette deuxième catégorie de sommets est donc en manque de flot à envoyer.

En gardant cette idée en tête, nous pouvons pointer le problème que pose l'arc bd : c'est un arc saturé qui transporte du flot depuis un sommet en manque vers un sommet en abondance. Cela semble contre-productif. Pour pouvoir contrer l'existence de tels arcs, il faudrait pouvoir faire diminuer le flot sur des arcs faisant circuler une quantité strictement positive de flot. C'est l'idée de l'algorithme de Ford et Fulkerson : plutôt que de prendre un st-chemin passant par des arcs

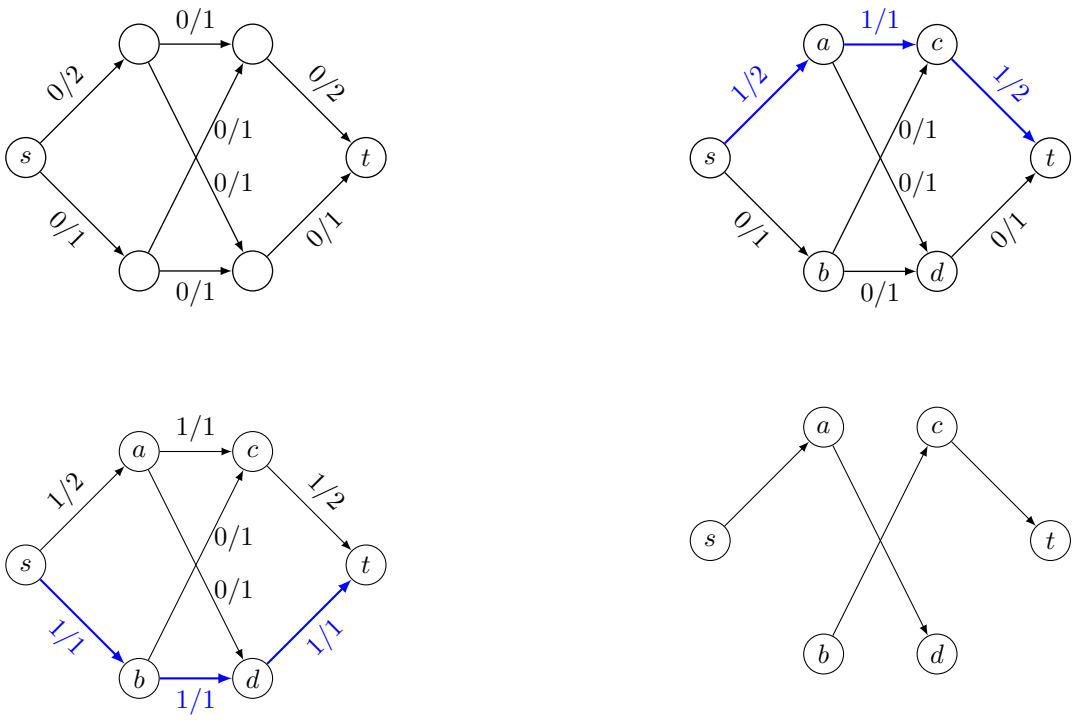


FIGURE 2.12 – L'algorithme naïf de calcul d'un flot : nous trouvons deux st-chemins permettant d'augmenter la valeur du flot d'une unité chacun. Puis il n'existe plus de st-chemins ne passant que par des arêtes non-saturées, comme le montre le graphe en bas à droite.

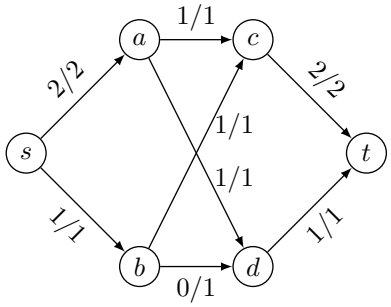


FIGURE 2.13 – Un flot maximum pour le graphe de la Figure 2.12. Il a pour valeur 3.

non-saturés, nous allons permettre au *st*-chemin d'utiliser des arcs en sens inverse, sur lesquels le flot sera diminué plutôt qu'augmenté.

Définition 2.5.4. *Soit f un flot sur le graphe G, c. Le* graphe résiduel G_f *du flot f est le graphe sur les sommets $V(G)$, et ayant pour arcs :*
- *les arcs* avant $\overrightarrow{e}$ *pour tout $e \in E(G)$ tel que $f(e) < c(e)$, avec $src(\overrightarrow{e}) = src(e)$ et $dst(\overrightarrow{e}) = dst(e)$;*
- *les arcs* arrière $\overleftarrow{e}$ *pour tout $e \in E(G)$ tel que $f(e) > 0$, avec $src(\overleftarrow{e}) = dst(e)$ et $dst(\overleftarrow{e}) = src(e)$.*

La capacité résiduelle c_f *d'un arc de G_f est définie par :*
- $c_f(\overrightarrow{e}) = c(e) - f(e)$,
- $c_f(\overleftarrow{e}) = f(e)$.

La capacité résiduelle définit donc sur un arc avant de combien on peut augmenter le flot sans violer la loi de capacité, et sur un arc arrière de combien on peut le diminuer. Par définition du graphe résiduel, la capacité résiduelle d'un arc du graphe résiduel est toujours strictement positive. Le graphe résiduel est en effet une façon de représenter quelles sont les libertés qui nous sont données par le flot actuel vis-à-vis de la loi de capacité : sur quels arcs nous pouvons modifier le flot, et dans quelle direction. Notez bien qu'un arc de G peut donner lieu à un arc avant et un arc arrière simultanément.

Nous allons donc chercher un *st*-chemin dans le graphe résiduel du flot actuel, et augmenter le flot sur les arcs avants de ce chemin, diminuer le flots sur ses arcs arrières. Ceci est justifié par le prochain lemme.

Lemme 2.5.2. *Soit P un st-chemin de G_f ne contenant pas de cycle, alors f' est un flot, avec :*
- $f'(e) = f(e) + \Delta(P)$ *si* $\overrightarrow{e} \in P$,
- $f'(e) = f(e) - \Delta(P)$ *si* $\overleftarrow{e} \in P$,
- $f'(e) = f(e)$ *dans les autres cas.*

et $\Delta(P) = \min_{\overline{e} \in P} c_f(\overline{e}) > 0$.

Démonstration. Vérifions d'abord la loi de capacité. Si $\overrightarrow{e} \in P$, alors $0 < f'(e) = f(e) + \Delta(P) \leq f(e) + c(e) - f(e) = c(e)$. Si $\overleftarrow{e} \in P$, alors $c(e) > f'(e) = f(e) - \Delta(P) \geq f(e) - f(e) = 0$. Enfin sinon, $0 \leq f'(e) = f(e) \leq c(e)$.

Vérifions maintenant la loi de conservation. Soit $u \notin \{s, t\}$ un sommet traversé par P, soit $\overline{e_1}$ et $\overline{e_2}$ les deux arcs successifs de P incidents à u : $\mathrm{dst}(\overline{e_1}) = u = \mathrm{src}(\overline{e_2})$.
- si $\overrightarrow{e_1}$ et $\overrightarrow{e_2}$, les capacités entrantes et sortantes de u sont augmentées de $\Delta(P)$ chacune.
- si $\overleftarrow{e_1}$ et $\overleftarrow{e_2}$, les capacités entrantes et sortantes de u sont diminuées de $\Delta(P)$ chacune.
- si $\overleftarrow{e_1}$ et $\overrightarrow{e_2}$, la capacité sortante de u est augmentée de $-\Delta(P) + \Delta(P) = 0$, la capacité entrante ne change pas.
- sinon $\overrightarrow{e_1}$ et $\overleftarrow{e_2}$, la capacité entrante est augmentée de $\Delta(P) - \Delta(P) = 0$, la capacité sortante ne change pas.

Dans tous les cas, la loi de conservation est maintenue. □

Motivé par ce lemme, nous introduisons la définition :

Définition 2.5.5. *Pour un flot f sur le graphe G, c, nous appelons* chemin augmentant *tout st-chemin du graphe résiduel G_f.*

Nous pouvons donc maintenant finir le calcul du flot maximum du graphe de la Figure 2.12 : le graphe résiduel contient un *st*-chemin utilisant l'arc $\overleftarrow{bd}$ comme arc arrière, justement l'arc qui nous semblait douteux. Nous augmentons le flot sur les arcs avant, et diminuant le flot sur les arcs arrière du chemin augmentant (Figure 2.14).

L'algorithme de Ford-Fulkerson peut donc se résumer ainsi :
- partir du flot nul,
- tant qu'il existe une chemin augmentant pour le flot actuel, augmenter le flot selon ce chemin,

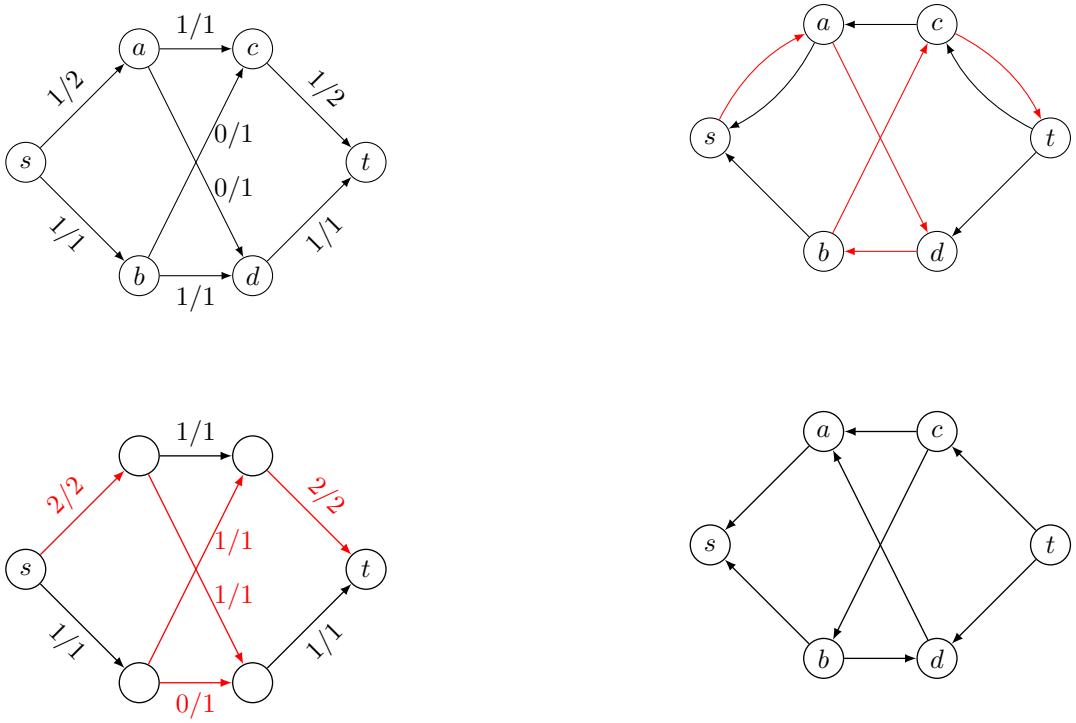


FIGURE 2.14 – Utilisation d'un arc arrière dans le calcul d'un chemin. Un flot sous-optimal en haut à gauche. En haut à droite le graphe résiduel pour ce flot sous-optimal. En bas à gauche le flot après augmentation, et son graphe résiduel en bas à droite. Il n'existe plus de st-chemin dans le dernier graphe résiduel, l'algorithme est terminé.

— s'il n'existe pas de chemin augmentant, le flot actuel est optimal.

Le graphe résiduel est composé :

— des arcs non-saturés (en avant), puisqu'on peut augmenter leur flot,

— des arcs ayant un flot strictement positif (en arrière), puisqu'on peut diminuer leur flot.

Implémentation. Nous proposons une implémentation de l'algorithme en Figure 2.16. Considérant que l'algorithme procède essentiellement à des parcours du graphe résiduel pour déterminer un chemin augmentant, il est préférable d'utiliser une représentation par liste d'incidence pour cet algorithme. À noter qu'il est nécessaire de coder à la fois les listes d'incidence entrante et sortante de chaque sommet.

Correction et complexité. Nous commençons par démontrer que cet algorithme calcule bien un flot maximum. Pour cela, il nous faut démontrer la maximalité du flot obtenu. Nous utilisons le concept de st-coupe pour cela :

Définition 2.5.6. *Une st-coupe d'un graphe orienté G (avec $s, t \in V(G)$) est un ensemble d'arcs $F \subset E(G)$ tel qu'il existe un ensemble de sommets $U \subseteq V(G)$ contenant s mais pas t, tel que les arcs de F sont les arcs ayant une et une seule extrémité dans U : on note $F = \delta(U)$. On parle de coupe orienté lorsqu'on considère l'ensemble des arcs sortants de l'ensemble de sommets u : $\delta^+(U) := \{(u, v) \in E(G) \mid u \in U, v \notin U\}$.*

Lemme 2.5.3. *Soit $U \subset V(G)$ un ensemble de sommets tel que $s \in U$ mais $t \notin U$. Alors la valeur maximum d'un st-flot est au plus :*

$$\sum_{e \in \delta^+(U)} c(e)$$

On appelle cette quantité la capacité de la coupe $\delta^+(U)$.

Démonstration. Nous nous rappelons que la valeur d'un st-flot est l'excès en t, ou la négation de l'excès en s. Mais nous avons :

$$\begin{aligned}
\text{excès}(t) &= \sum_{v \in V(G) \setminus U} \text{excès}(v) \\
&= \sum_{v \in V(G) \setminus U} \left(\sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e) \right) \\
&= \sum_{e \in \delta^+(U)} f(e) - \sum_{e \in \delta^-(U)} f(e) \\
&\leq \sum_{e \in \delta^+(U)} c(e) \qquad (2.1)
\end{aligned}$$

où la deuxième égalité est par définition de l'excès, la troisième car tous les arcs entre deux sommets de $V(G) \setminus U$ sont comptés une fois positivement et une fois négativement, et l'inégalité vient de la loi de capacité. □

Il suffit donc de montrer maintenant qu'il existe une coupe dont la capacité sortante est égale à la valeur du flot que trouve l'algorithme.

Théorème 2.5.1 (Max-flot Min-coupe)**.** *Pour tous G, c, s, t, la valeur du flot maximum est égale à la valeur de la st-coupe minimum. De plus, l'algorithme de Ford-Fulkerson est correct.*

Démonstration. Nous admettons pour l'instant que l'algorithme de Ford-Fulkerson termine (nous le prouverons plus tard).

Soit R l'ensemble des sommets accessibles depuis s dans le graphe résiduel du flot f trouvé par l'algorithme de Ford-Fulkerson. Alors pour tout arc $e \in \delta^+(R)$ sortant de R, $f(e) = c(e)$ (sinon e

```
1  type orientation = Avant ou bien Arriere
2  type arc_résiduel = (arc, orientation)
3
4  // Les fonctions origine et destination du graphe résiduel :
5  fonction origine(arc_résiduel (arc, direction)) : sommet :=
6    si direction = Avant alors retourner queue(arc) sinon retourner tête(arc)
7  fonction destination(arc_résiduel (arc, direction)) : sommet :=
8    si direction = Arriere alors retourner queue(arc) sinon retourner tete(arc)
9
10 // La capacité résiduelle d'un arc :
11 fonction capacité_résiduelle(arc_résiduel (arc, direction)) : flottant :=
12   si direction = Avant alors retourner capacité(arc) − flot[arc]
13   sinon retourner flot[arc]
14
15 // Augmentation du flot sur un arc résiduel :
16 fonction augmente(arc_résiduel (arc, direction), flottant delta) : sommet :=
17   si direction = Avant alors flot[arc] ← flot[arc] + delta
18   sinon flot[arc] ← flot[arc] − delta
19
20 // Le graphe résiduel :
21 fonction construis_graphe_résiduel(graphe g) : graphe :=
22   soit résiduel = graphe_vide()
23   pour tout sommet ∈ sommets(g) faire
24     ajoute_sommet(résiduel, sommet)
25     pour tout arc ∈ arcs_sortants(g, v) faire
26       si flot[arc] < capacité(arc) alors ajoute_arc(résiduel, (arc, Avant))
27     pour tout arc ∈ arcs_entrants(g, v) faire
28       si flot[arc] > 0 alors ajoute_arc(résiduel, (arc, Arriere))
29   retourner résiduel
30
31 // Construire le chemin depuis le résultat du parcours :
32 fonction construis_chemin(tableau d'arc_résiduels prédécesseur, sommet dernier_sommet)
33               : liste d'arc_résiduels :=
34   soit chemin := liste_vide
35   soit premier_sommet := dernier_sommet
36   tant que prédécesseur[premier_sommet] ≠ ⊥ faire
37     chemin ← insère(prédécesseur[premier_sommet], chemin)
38     premier_sommet ← origine(prédécesseur(premier_sommet))
39   retourner chemin
```

FIGURE 2.15 – Les fonctions et types utiles pour l'algorithme de Ford-Fulkerson. Nous supposons que les valeurs de flots sont contenus dans un tableau indicé par les arcs.

```
40  // Algorithme de Ford-Fulkerson :
41  fonction ford_fulkerson(graphe g, sommet source, sommet puits) : void :=
42    pour_tout arc ∈ arcs(g) faire flot[arc] ← 0
43    tant que true faire
44      soit prédécesseur := parcours(construis_graphe_résiduel(g), source)
45      si prédécesseur[puits] = ⊥ alors retourner
46      soit chemin_augmentant := construis_chemin(prédecesseur, puits)
47      soit delta := min{capacité_résiduelle(arc_résiduel) | arc_résiduel ∈ chemin_augmentant}
48      pour tout arc_résiduel ∈ chemin_augmentant faire
49        augmente(arc_résiduel, delta)
```

FIGURE 2.16 – Le cœur de l'algorithme de Ford-Fulkerson. Nous utilisons, en plus des fonctions utiles décrites en Figure 2.15, d'un algorithme de parcours de graphes, tel que celui en Figure 2.3

donnerait un arc avant dans G_f, et la tête de e serait accessible depuis R, contradiction), et pour tout arc $e \in \delta^-(R)$ entrant de R, $f(e) = 0$ (sinon e donnerait un arc arrière de G_f, et la queue de e serait alors accessible depuis R). Alors, l'inégalité de la ligne (2.1) est en fait une égalité. Donc l'excès en t est égal à la valeur de la coupe $\delta^+(R)$. □

Attardons-nous maintenant sur le problème de la terminaison et de la complexité de cet algorithme. La mauvaise nouvelle est que sans autre hypothèse, l'algorithme de Ford-Fulkerson, dans toute sa généralité, ne termine pas nécessairement. Uri Zwick a ainsi construit un graphe à 6 sommets et 8 arcs, dont un arc ayant une capacité irrationnelle, tel que l'algorithme, s'il fait un choix précis de chemins augmentants à chaque itération, ne parvient jamais au flot maximum : l'augmentation à chaque étape devient plus petite, convergeant vers 0.

Heureusement, il existe des solutions. D'abord, nous pouvons interdire les capacités irrationnelles. Si les capacités sont rationnelles, nous pouvons multiplier ces capacités par une constante strictement positive bien choisie pour obtenir des capacités entières, et cela multiplie le flot par la même constante, ainsi l'instance entière obtenue est équivalente à l'instance rationnelle.

Proposition 2.5.2. *L'algorithme de Ford-Fulkerson termine sur les instances à capacité entière (ou rationnelle).*

Démonstration. On montre par induction que pour tout n le flot obtenu après n augmentations est entier. Le flot initial est entier ($n = 0$). Soit $n \geq 0$, supposons que le n^e flot f_n est entier. Alors les capacités résiduelles sont entières, donc l'augmentation est aussi entière, et au moins égale à 1, tout comme f_{n+1}.

Comme chaque itération augmente la valeur du flot, le nombre d'itération est borné par la valeur de la capacité minimum d'une t-coupe de G, qui est aussi un entier. □

Le nombre d'itérations est ainsi borné par la valeur du flot maximum. C'est une borne de piètre qualité, car même un très petit graphe peut avoir un très grand flot. Et effectivement, l'exemple de la Figure 2.17 montre que ce n'est pas juste un problème avec l'analyse mathématique, mais que sans autre hypothèse l'algorithme peut être excessivement lent.

Nous utilisons donc une méthode supplémentaire pour drastiquement réduire la complexité asymptotique de l'algorithme de Ford-Fulkerson. Jusqu'à présent, nous n'avons jamais précisé comment sont choisis les chemins augmentants dans le graphe résiduel. Certainement nous voulons en pratique utiliser un algorithme de parcours, si possible en largeur ou en profondeur car ils ont de très bonnes complexités, et trouver ces chemins représente l'essentiel des calculs effectués par l'algorithme. Edmonds et Karp, et indépendemment Dinitz, ont proposé d'utiliser un parcours en largeur, et ainsi de choisir un plus court st-chemin en nombre d'arcs. Nous montrons que cela garantit une complexité qui ne dépend que du nombre de sommets et du nombre d'arcs, et pas de la valeur du flot.

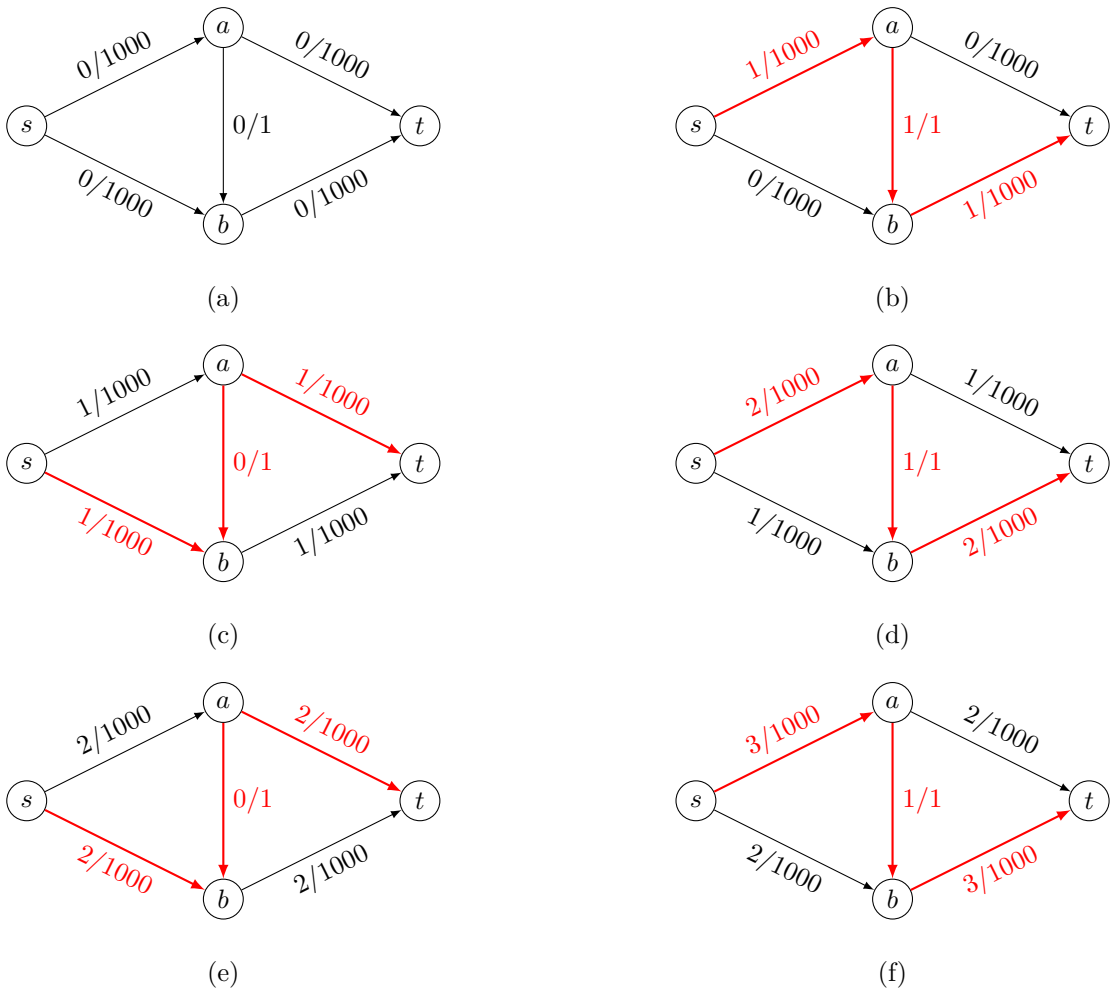


FIGURE 2.17 – Il faut 2000 itérations à l'algorithme pour terminer s'il choisit systématiquement un chemin passant par l'arc vertical dans cet exemple.

Le principe de l'analyse repose sur l'observation que la longueur des chemins dans le graphe résiduel depuis s ne fait que croître à chaque itération de l'algorithme. Comme la distance maximum d'un sommet accessible est borné par le nombre de sommets (un plus court chemin passe au plus une fois par chaque sommet), il suffira de montrer que les distances augmentent régulièrement pour obtenir une borne sur le nombre d'itérations.

Rappelons que la distance entre $u \in V$ et $v \in V$ dans un graphe $G = (V, E)$ est donné par $d_G(u, v) := \min_{P\ uv\text{-chemin}} |P|$, et $d_G(u, v) := +\infty$ si v n'est pas accessible depuis u.

Lemme 2.5.4. *Soit $G = (V, E)$ un graphe, c une fonction de capacité et f un flot pour G, c. Soit f' un flot de G, c obtenu depuis f par l'augmentation sur un chemin de longueur minimum du graphe résiduel G_f. Alors pour tout sommet $v \in V$, $d_{G_{f'}}(s, v) \geq d_{G_f}(s, v)$.*

Démonstration. Quel est l'effet d'une augmentation sur le graphe résiduel ?

(i) si un arc avant est augmenté jusqu'à sa capacité, il disparait du graphe résiduel.

(ii) si un arc arrière est diminué jusqu'à zéro, il disparait aussi du graphe résiduel.

(iii) si un arc avant est augmenté depuis la valeur 0, l'arc arrière correspondant est ajouté au graphe.

(iv) si un arc arrière est diminué depuis sa capacité maximum, l'arc avant correspondant est ajouté au graphe.

Décomposons ces effets en deux parties. D'abord appliquons seulement les effets (iii) et (iv) pour obtenir un graphe intermédiaire entre G_f et $G_{f'}$. (iii) et (iv) rajoutent tous deux des arcs uv dont la queue u est plus loin de s que la tête v, en effet puisque le chemin augmentant est un plus court chemin $d_{G_f}(s, u) = d_{G_f}(s, v) + 1$. Donc un chemin utilisant un tel arc ne peut pas être un plus court chemin, les distances dans ce graphe intermédiaire sont donc les mêmes que dans G_f.

Ensuite, appliquons les effets (i) et (ii) qui suppriment des arcs du graphe intermédiaire. Supprimer des arcs ne peut pas créer de chemins plus courts que ceux déjà existant, donc les distances augmentent. Donc les distances dans $G_{f'}$ sont au moins égales aux distances dans G_f. □

Lemme 2.5.5. *Dénotons f_k le flot obtenu à l'itération k, et d_k la fonction de distance dans le graphe résiduel de f_k. Soit uv un arc de G_{f_i} qui n'apparait pas dans $G_{f_{i+1}}$ (donc uv est saturé lors de l'itération $i + 1$), et soit $j > i + 1$ minimum tel que uv réapparait dans G_{f_j}, alors $d_j(s, u) \geq d_i(s, u) + 2$.*

Démonstration.

$$d_j(s, u) = d_j(s, v) + 1 \geq d_i(s, v) + 1 = d_i(s, u) + 2$$

La première égalité est induite par le fait que vu est dans le chemin augmentant de l'itération j, l'inégalité centrale provient du Lemme 2.5.4, et la dernière égalité est impliqué par la présence de uv dans le chemin augmentant de l'itération $i + 1$. □

Ce lemme dit que chaque fois qu'un arc disparait puis réapparait dans le graphe résiduel, la distance de sa queue depuis la source a strictement augmenté.

Lemme 2.5.6. *Le nombre d'itérations nécessaire à l'algorithme de Ford-Fulkerson dans sa variante d'augmenter les plus courts chemins est au plus $O(mn)$.*

Démonstration. Chaque itération sature au moins une arête du chemin augmentant, donc fait disparaitre un arc uv du graphe résiduel. Puisque la distance d'un sommet depuis s est soit inférieure à n, soit infinie, et en vertu des Lemmes 2.5.4 et 2.5.5, un arc peut disparaître (et donc être saturé) au plus $n/2$ fois. Chaque arc de G donnant lieu à un arc avant et un arc arrière, ceci borne le nombre d'itérations à $m \times 2 \times (n/2) = mn$ (nombre d'arcs fois le nombre de saturations possibles) □

Théorème 2.5.2. *L'algorithme de Ford-Fulkerson peut être implémenté avec une complexité asymptotique $O(|V|.|E|^2)$.*

Démonstration. Il suffit d'ajouter au Lemme 2.5.6 que chaque itération a la complexité d'un parcours en profondeur, c'est-à-dire $O(|E| + |V|)$. □

Notons que cette fois nous n'avons même pas supposé que les capacités sont entières ou rationnelles. Le choix du plus court chemin garantit une terminaison rapide de l'algorithme même en présence de très grandes capacités ou de capacités irrationnelles.

Un exemple complet Nous présentons une exécution de l'algorithme sur un exemple un peu plus gros, en Figures 2.18, 2.19 et 2.20. Chaque ligne présente sur la gauche le flot actuel, avec en rouge les arcs ayant été augmenté lors de l'itération précédente, et sur la droite le graphe résiduel associé, avec en rouge un plus court st-chemin. L'algorithme utilise l'arc arrière eb à deux reprises pour les chemins des graphes résiduels (j) et (k), ce qui fait baisser le flot sur l'arc be.

En (n), il n'existe pas de st-chemin. Ceci marque la fin de l'algorithme. Les sommets accessibles dans le dernier graphe résiduel induisent une coupe de capacité minimum. Tous les arcs sortants de cette coupe sont saturés de flot (arcs verts en (o)), les arcs entrants n'ont plus de flot (arcs pointillés en (o)). La somme des capacités des arcs verts est 16, ce qui est aussi la valeur de flot maximum, conformément au Théorème 2.5.1 max-flot min-coupe.

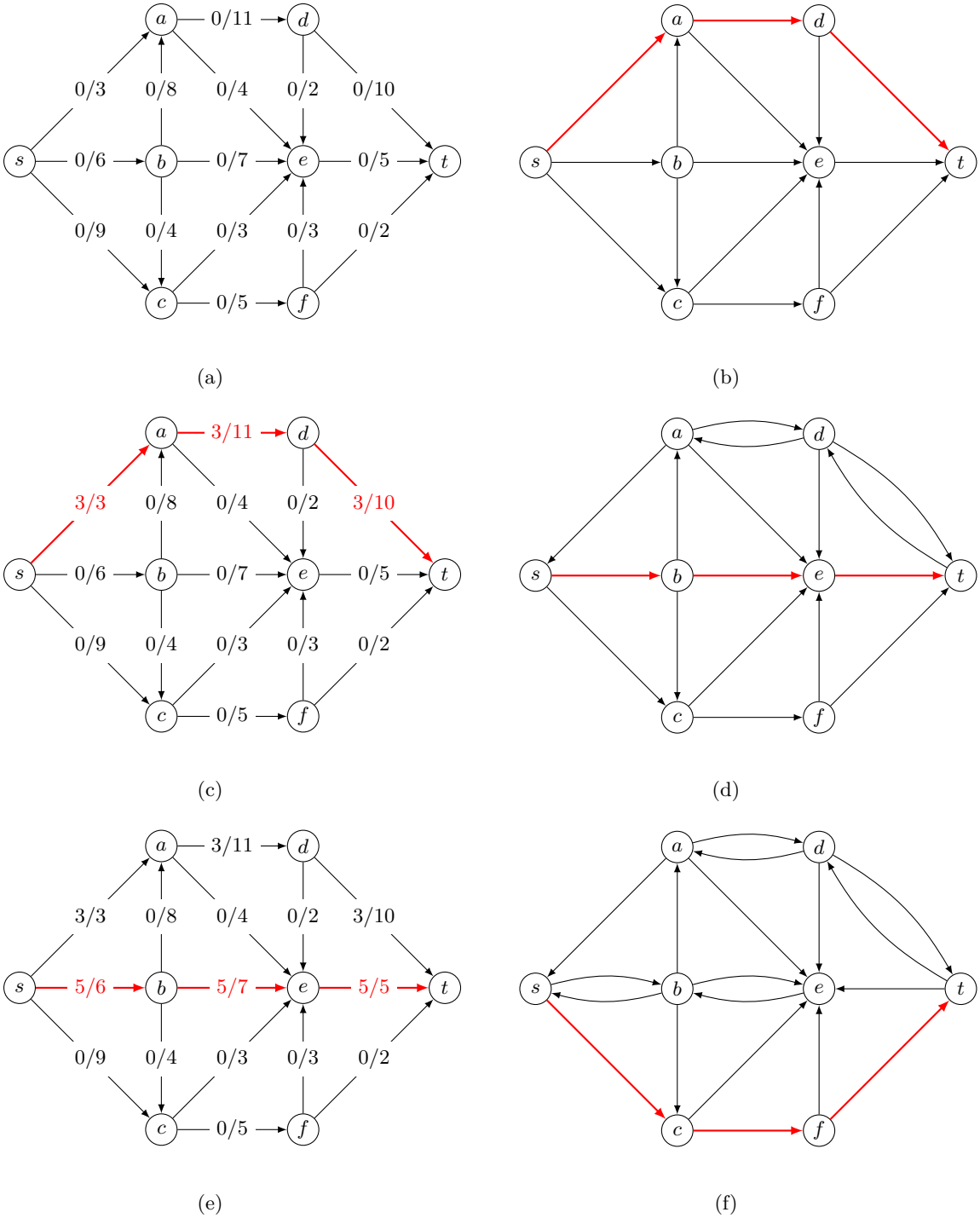


FIGURE 2.18 – Exécution de l'algorithme de Ford-Fulkerson, partie 1.

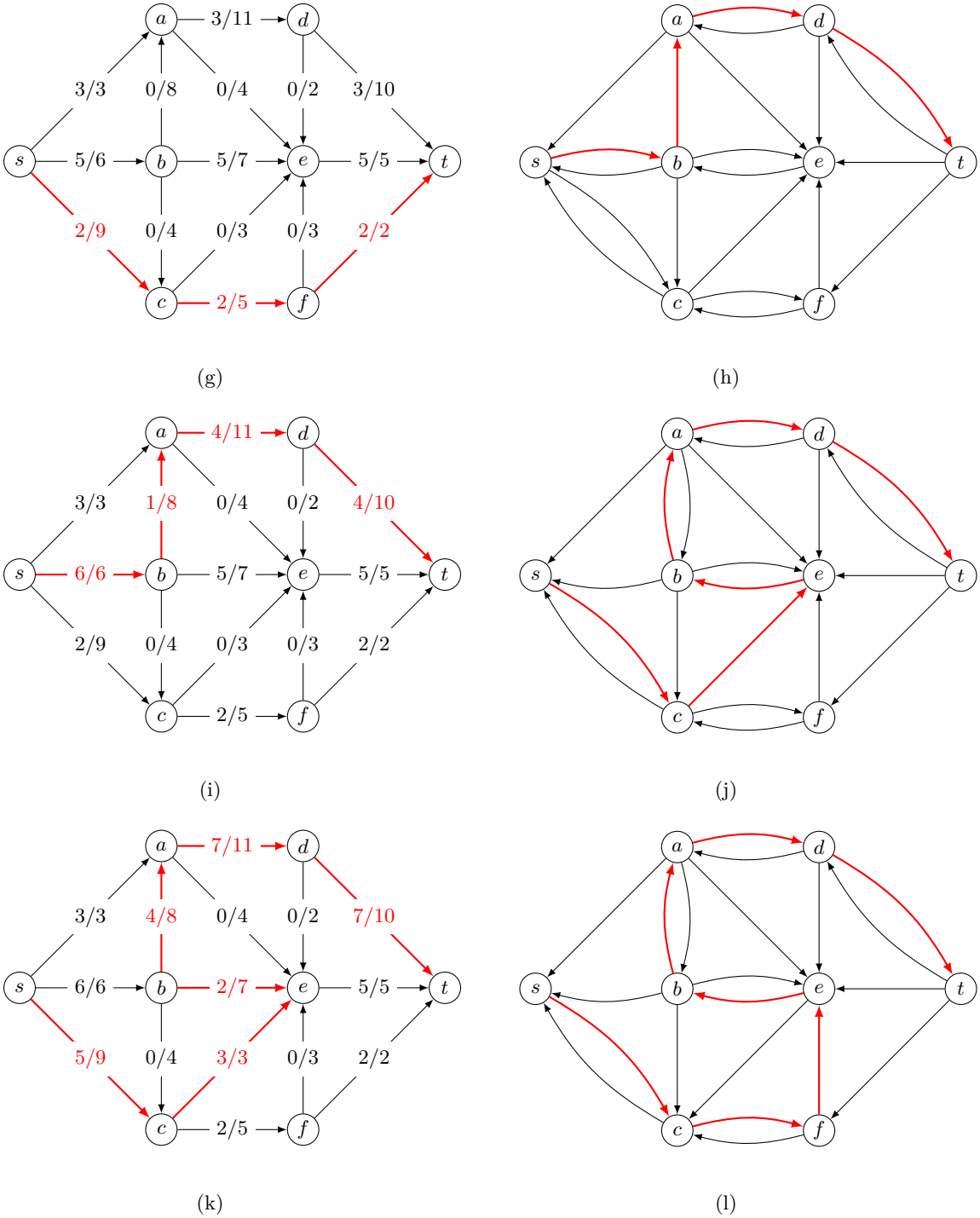
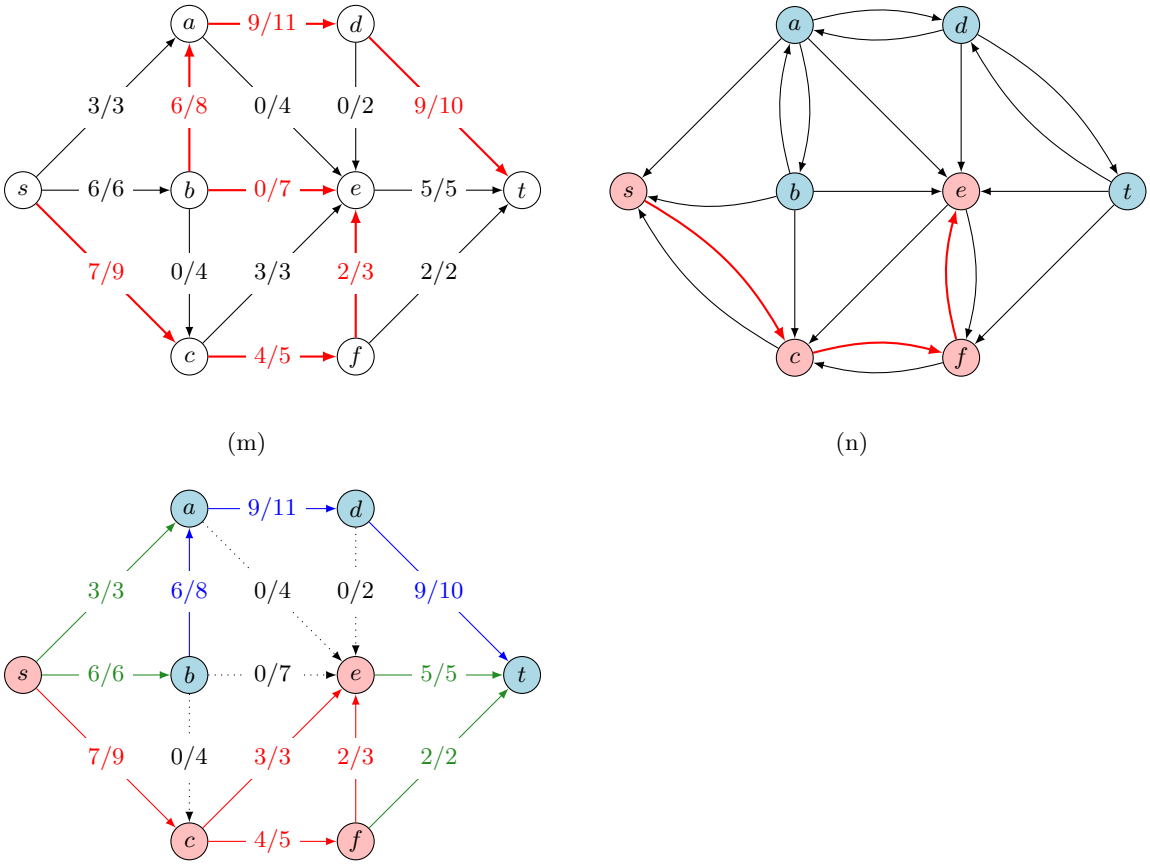


FIGURE 2.19 – Exécution de l'algorithme de Ford-Fulkerson, partie 2.



(m) (n)

FIGURE 2.20 – Exécution de l'algorithme de Ford-Fulkerson, partie 3.

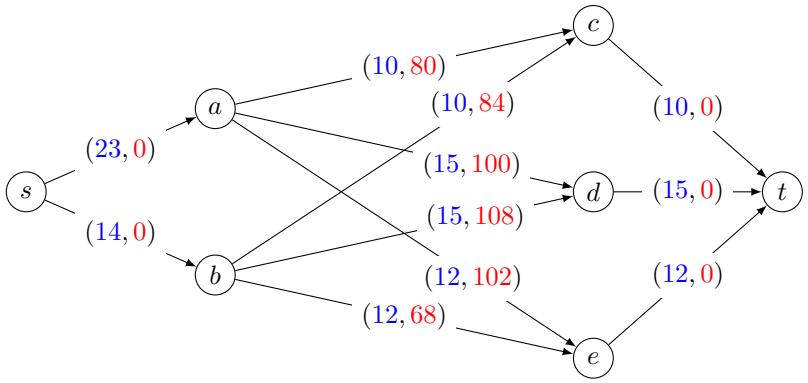


FIGURE 2.21 – Un réseau avec capacités (en bleu) et coûts (en rouge)

2.6 Extensions de flot maximum

2.6.1 Flot de coût minimum

Une extension intéressante du problème de flots consiste à considérer un réseau avec des capacités, mais aussi des coûts sur les arcs décrits par une fonction $q\colon E(G) \to \mathbb{Z}$. La fonction q décrit un coût par unité de flot : si $f(e)$ unité de flot passe à travers l'arc e, le coût associé à cet arc est $q(e) \times f(e)$. Un exemple de réseau avec coûts et capacités est donné en figure 2.21.

Définition 2.6.1. *Dans un graphe G muni de capacités $c\colon E(G) \to \mathbb{N}$ et de coûts $q\colon E(G) \to \mathbb{Z}$, le coût d'un flot $f\colon E(G) \to \mathbb{R}^+$ est défini par*

$$\text{cout}(f) = \sum_{e \in E} q(e) \times f(e)$$

Le problème du st-flot de coût minimum consiste alors à trouver un flot f de coût minimum (sans contrainte de maximisation de sa valeur) de valeur ϕ fixée.

Dans le réseau de la figure 2.21, si on abstrait les coûts, il est aisé de voir que la valeur du flot maximal est égal à 37. On cherche alors un flot de coût minimum réalisant cette valeur 37.

Il est facile de modifier l'algorithme de Ford-Fulkerson afin qu'il calcule un flot de coût minimum, plutôt qu'un flot (de valeur) maximum. Tout d'abord, on modifie la définition du graphe résiduel afin qu'il prenne en compte les coûts des arcs, plutôt que leur capacité. Le graphe résiduel G_f associé à un flot f est donc un graphe avec des coûts q_f, dont les sommets sont $V(G)$ et qui contient les arcs :

- (u, v) si $(u, v) \in E(G)$ n'est pas saturé par f (c'est-à-dire $f(e) < c(e)$), de coût $q_f(u, v) = q(u, v)$;
- (v, u) si $(u, v) \in E(G)$ porte un flot strictement positif (c'est-à-dire $f(u, v) > 0$)), de coût $q_f(v, u) = -q(u, v)$.

Comme avant, un arc (u, v) peut donc engendrer deux arcs dans le graphe résiduel : un arc avant (u, v) et un arc retour (v, u).

Plutôt que de chercher un plus court chemin augmentant en terme du nombre d'arcs du chemin, on cherche un plus court chemin augmentant en terme du coût, c'est-à-dire un plus court chemin dans le graphe résiduel G_f entre la source s et le puits t. Pour cela, on peut utiliser un algorithme de calcul de plus court chemin dans un graphe avec des poids négatifs (par exemple l'algorithme de Bellman-Ford ou Floyd-Warshall). On augmente alors le flot dans le réseau d'origine en utilisant le chemin augmentant correspondant : chaque arc est augmenté du minimum entre la capacité résiduelle minimale et la différence entre ϕ et la valeur courante du flot (puisqu'on cherche à obtenir un flot de valeur ϕ à la fin de l'algorithme).

L'exécution de cet algorithme, qu'on appelle algorithme de Busacker-Gowen, sur le réseau de la figure 2.21 est donné en figure 2.22.

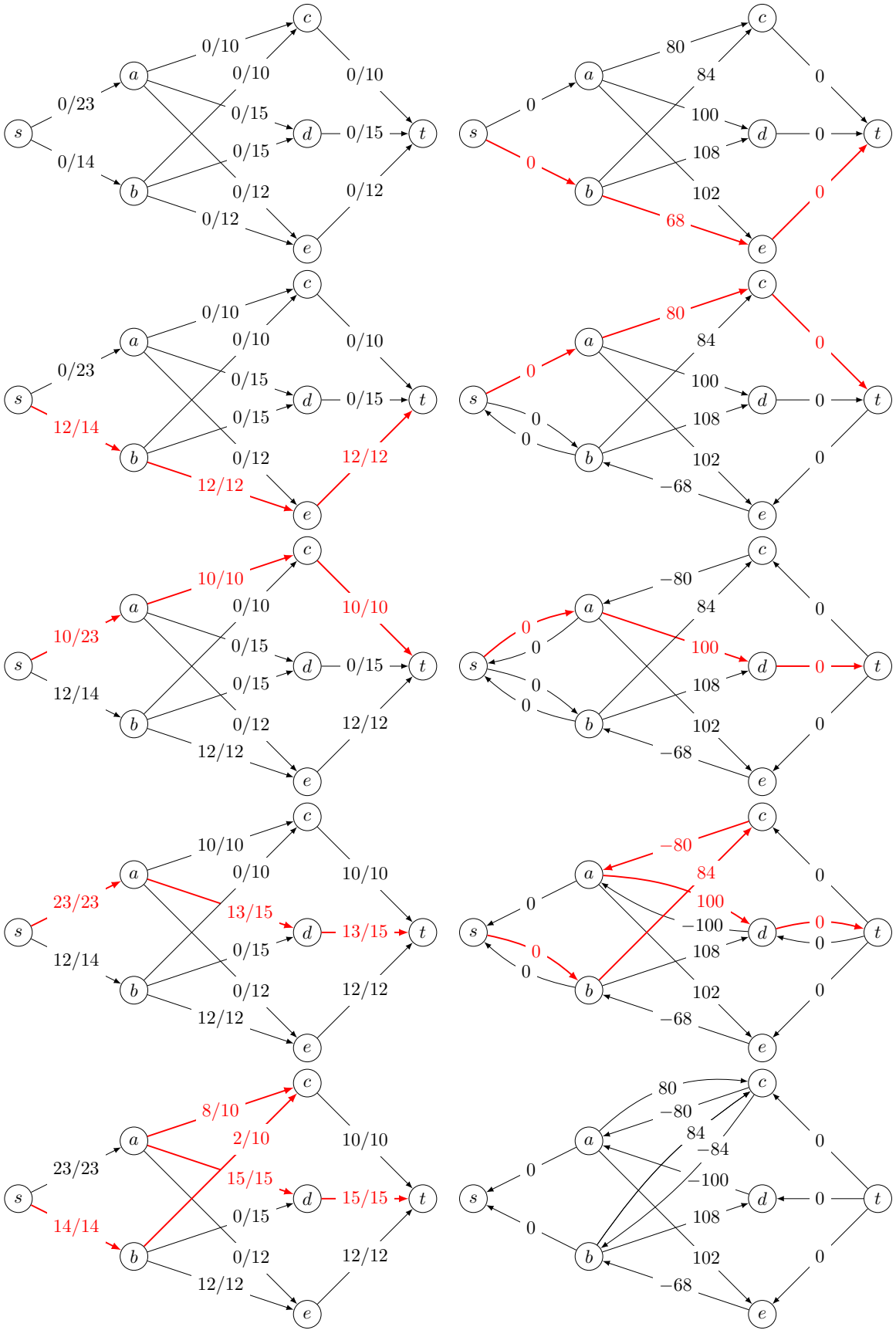


FIGURE 2.22 – À gauche le réseau étiqueté par le flot courant et les capacités (on ne représente donc pas les coûts) ; à droite, le graphe résiduel avec les coûts et le chemin augmentant trouvé par l'algorithme de plus court chemin

Étudions la correction, la terminaison et la complexité de l'algorithme de Busacker-Gowen.

Proposition 2.6.1. *Si un flot f a valeur w et est de coût minimum parmi tous les flots de valeur w, alors tous les cycles de G_f ont un coût (égal à la somme des coûts des arcs du cycle) positif ou nul.*

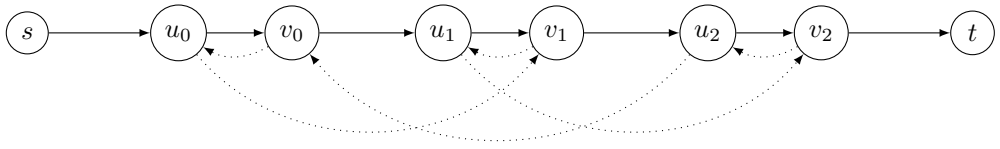
Démonstration. Supposons qu'il existe un cycle C de coût strictement négatif dans G_f. On peut alors faire circuler davantage de flot sur C, en augmentant d'une unité le flot des arcs avant du circuit, et en le diminuant d'une unité pour les arcs arrière. Ceci ne change pas la valeur du flot, mais ajoute au coût du flot le coût de C, strictement négatif. Ceci contredit donc la minimalité du coût de f. □

Notons que la réciproque de cette proposition est vraie aussi, mais plus délicate à démontrer. Comme corollaire, on obtient l'invariant suivant pour l'algorithme de Busacker-Gowen.

Corollaire 2.6.1. *À chacune des itérations de l'algorithme de Busacker-Gowen, le flot courant est de coût minimum parmi tous ceux de même valeur.*

Démonstration. La preuve se fait pas récurrence sur la valeur du flot. Pour le cas de base, la propriété est triviale initialement lorsque le flot a valeur 0.

Supposons que l'on ait un flot f de coût minimum au début d'une itération et que l'augmentation de flot sur un chemin P de coût minimum de G_f conduise à un flot f' non optimal en fin d'itération. D'après la proposition 2.6.1, il existe un cycle C de coût strictement négatif dans le graphe $G_{f'}$. Ce cycle n'existe pas dans G_f sinon le flot f ne serait pas de coût minimum. Ce cycle emprunte donc un arc créé par l'augmentation de flot selon P, donc un arc (v, u) qui était un arc (u, v) sur le chemin P (seule façon de faire apparaître un cycle). Notons $\{(u_i, v_i) \mid 1 \leq i \leq k\}$ l'ensemble de ces arcs (u, v) de P qui apparaissent dans C sous la forme (v, u). Le cycle C parcourt donc les arcs (v_i, u_i) dans un certain ordre, relié par des chemins d'un u_i à un v_j. Dans l'exemple ci-dessous, le chemin P est tracé en traits pleins, et le cycle C en traits pointillés.



Soit P' le st-chemin obtenu en empruntant les parties du st-chemin de P hors des arcs (u_i, v_i) et du cycle C hors des arcs (v_i, u_i) : dans l'exemple ci-dessus, il s'agit du chemin passant par s, u_0, v_1, u_2, v_0, u_1, v_2 et t. Son poids est obtenu en ajoutant le poids de P et le poids de C, et en supprimant le poids des arcs (u_i, v_i) et (v_i, u_i). Puisque les poids des arcs (u_i, v_i) et (v_i, u_i) sont opposés, et que le poids de C est strictement négatif, le poids de P' est donc strictement inférieur au poids de P. On a donc trouvé un st-chemin dans G_f de poids strictement inférieur à P ce qui contredit le choix de P. □

Cela assure que si l'algorithme termine (lorsqu'on a trouvé un flot de la valeur ϕ désirée), il est correct. Supposons que les capacités sont entières, comme pour l'algorithme de Ford-Fulkerson. Notons $\alpha = \max_{e \in E(G)} q(u, v)$ la capacité maximale du réseau. La capacité de toute coupe est au plus $p = \alpha n$. On peut avoir p itérations si le poids de tous les plus courts chemins est de 1. Ceci prouve la terminaison de l'algorithme. En utilisant l'algorithme de Bellman-Ford pour calculer les plus courts chemins dans le graphe résiduel, de complexité $O(nm)$, on assure que l'algorithme de Busacker-Gowen a une complexité $O(n^2 m\alpha)$. Il a donc les mêmes défauts que l'algorithme de Ford-Fulkerson : Goldberg et Tarjan ont obtenu un algorithme plus efficace (en temps polynomial $O(m^3 n^2 \log n)$) qui recherche les cycles de coûts minimum dans le graphe résiduel, plutôt que les chemins de coûts minimum...

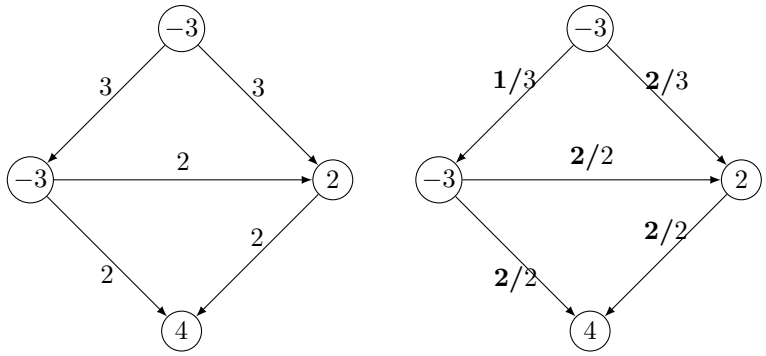


FIGURE 2.23 – Circulation avec demandes : à gauche le graphe avec des coûts sur les arcs et des demandes sur les sommets, à droite une circulation valide

2.6.2 Circulation avec demandes

Le problème de *circulation avec demandes* est une autre extension du problème de flot maximum où la source et le puits sont remplacés par des demandes $d_v \in \mathbb{Z}$ sur chacun des sommets $v \in V(G)$ du graphe :

— une demande $d_v > 0$ correspond à un puits v qui requiert un flot d'exactement d_v unités ;
— une demande $d_v < 0$ correspond à une source qui produit exactement $-d_v$ unités de flot ;
— une demande $d_v = 0$ correspond à un sommet classique.

On note S l'ensemble des sommets qui sont des sources et T l'ensemble des sommets qui sont des puits. Dans l'exemple de la figure 2.23, il y a deux sources (en haut et à gauche) et deux puits (à droite et en bas).

Définition 2.6.2. *Soit G un graphe orienté et $c\colon E(G) \to \mathbb{R}^+$ une fonction de capacité. Une circulation avec demandes $(d_v)_{v\in V(G)}$ est une fonction $f\colon E(G) \to \mathbb{R}^+$ telle que :*

— ***Loi de capacité*** $\forall e \in E\ f(e) \leq c(e)$;
— ***Loi de conservation*** $\forall v \in V$

$$\sum_{e\in\delta^+(v)} f(e) - \sum_{e\in\delta^-(v)} f(e) = d_v$$

À droite de la figure 2.23, on peut voir une circulation avec demandes.

Lemme 2.6.1. *S'il existe une circulation avec demandes $(d_v)_{v\in V(G)}$, alors $\sum_{v\in V(G)} d_v = 0$.*

Démonstration. Soit f une circulation valide. Alors $d_v = \sum_{e\in\delta^+(v)} f(e) - \sum_{e\in\delta^-(v)} f(e)$. En sommant sur tous les sommets, on obtient :

$$\sum_{v\in V(G)} d_v = \sum_{v} \sum_{e\in\delta^+(v)} f(e) - \sum_{v} \sum_{e\in\delta^-(v)} f(e)$$

Le flot sur chaque arc est donc compté deux fois, une fois positivement et une fois négativement, de sorte que $\sum_{v\in V(G)} d_v = 0$. □

Cela implique qu'il existe une solution si et seulement si

$$D = \sum_{v,d_v>0} d_v = -\sum_{v,d_v<0} d_v$$

Pour résoudre un tel problème de circulation avec demande, on le transforme en un problème de flot maximum classique :

1. On commence par vérifier que $\sum_{v,d_v>0} d_v = -\sum_{v,d_v<0} d_v$.

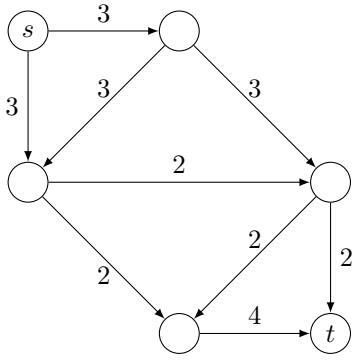


FIGURE 2.24 – Réseau G' avec une super-source s et un super-puits t

2. On crée une *super-source* s et on la connecte à tous les sommets v tels que $d_v < 0$, en fixant la capacité $c(s, v)$ du nouvel arc à $-d_v$.
3. On crée un *super-puits* t et on le connecte à tous les sommets v tels que $d_v > 0$, en fixant la capacité $c(v, t)$ du nouvel arc à d_v.
4. Dans le graphe G' ainsi obtenu, on calcule le flot maximum de s à t : s'il est égal à D alors il existe une circulation valide qu'on obtient en restreignant le flot au graphe G originel. Sinon, il n'existe pas de circulation valide.

Pour le réseau de la figure 2.23, le graphe G' obtenu en ajoutant la super-source et le super-puits est décrit en figure 2.24.

Théorème 2.6.1. *Il existe une circulation avec demandes (d_v) dans un graphe G munis de capacités positifs si et seulement si le st-flot maximum dans le graphe G' modifié a valeur D. Si toutes les capacités de G sont entières et qu'il existe une circulation avec demandes, alors il existe une circulation avec demandes à valeurs entières.*

Il est possible, comme pour les flots dans la section précédente, de rechercher une circulation avec demande de coût minimum : le même genre de transformation permet de transformer ce problème en la recherche d'un flot de coût minimum de valeur D.

2.6.3 Circulation avec demandes et bornes inférieures de capacité

Ajoutons désormais au problème de circulation avec demandes une borne inférieure $\ell(e) \in \mathbb{N}$ pour chaque arc $e \in E(G)$ sur la quantité de flot désirée : on assure que $\ell(e) \leq c(e)$ pour que le problème ne soit pas trivialement insoluble.

Définition 2.6.3. *Une circulation avec demandes (d_v) et bornes inférieures décrites par $\ell : E(G) \to \mathbb{N}$ est une circulation $f : E(G) \to \mathbb{R}^+$ vérifiant de plus que $\ell(e) \leq f(e)$ pour tout arc $e \in E(G)$.*

Un exemple de réseau de circulation avec demandes et bornes inférieurs est représenté en figure 2.25.

Naïvement, initialisons notre recherche de circulation par la fonction f qui assigne à chaque arc e la borne inférieure $\ell(e)$. Ce n'est évidemment pas toujours un flot, et cette fonction ne respecte pas toujours les demandes. Notons f_0 ce flot initial : $\forall e$, $f_0(e) = \ell(e)$. Pour tout sommet v, notons

$$L_v = \sum_{e \in \delta^-(v)} f_0(e) - \sum_{e \in \delta^+(v)} f_0(e)$$

Si $L_v = d_v$, on est content puisque le flot satisfait les demandes requises. Sinon, il faut régler le déséquilibre en v. Pour cela, transformons les demandes et les capacités : on pose $d'_u = d_u - L_u$ pour tous les sommets u, et $c'(e) = c(e) - \ell(e)$ pour tous les arcs e. Notons G' le nouveau réseau

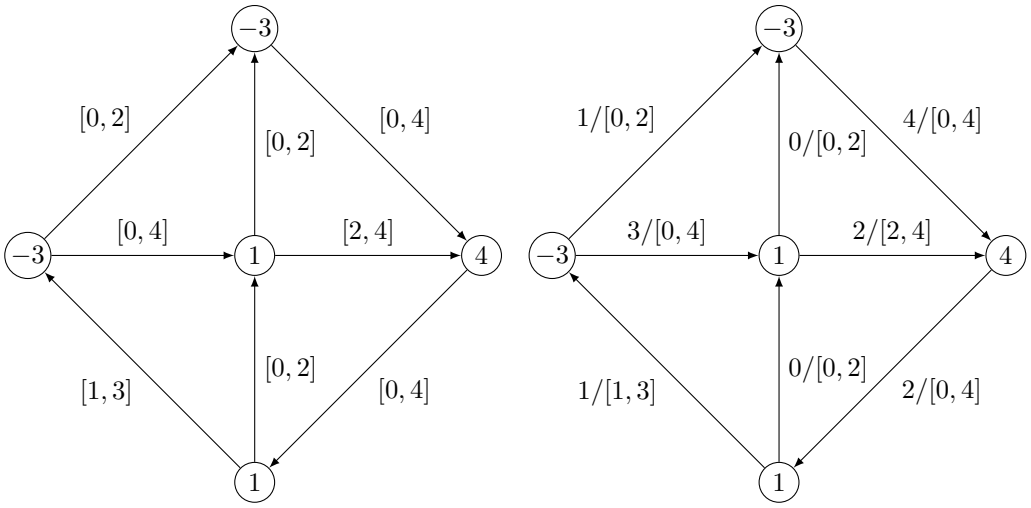


FIGURE 2.25 – Circulation avec demandes et bornes inférieures : à gauche, chaque arc est étiqueté par $[\ell(e), c(e)]$; à droite, une circulation valide

ainsi obtenu (sans bornes inférieures). Si on trouve une circulation f' dans G' qui satisfait ces nouvelles demandes, alors, clairement, le flot $f = f_0 + f'$ est une circulation valide qui satisfait de plus les bornes inférieures $\ell(e)$. En utilisant l'algorithme précédent, on résout donc ce nouveau problème.

Lemme 2.6.2. *Il existe une circulation avec demandes et bornes inférieures dans G si et seulement s'il existe une circulation avec demandes dans le nouveau réseau G'. Si toutes les demandes, capacités et bornes inférieures de G sont entières, et qu'il existe une circulation valide, alors il en existe une à valeurs entières.*

Démonstration. Soit f' une circulation valide dans G'. Soit $f = f_0 + f' = \ell + f'$. Clairement, puisque $f'(e) \geq 0$, f vérifie les bornes inférieures ℓ. De plus, pour tout sommet v,

$$\sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e) = \sum_{e \in \delta^-(v)} (f_0(e) + f'(e)) - \sum_{e \in \delta^+(v)} (f_0(e) + f'(e)) = L_v + (d_v - L_v) = d_v$$

La réciproque s'obtient similairement. □

Il est possible de définir également le problème de flot maximum avec bornes inférieures de capacité.

2.7 Recherche de plus courts chemins depuis une source (vu en L2)

Nous avons vu que le parcours en largeur permet de calculer pour tous les sommets accessibles depuis la source s un chemin de longueur minimale. Rappelons que nous avons défini la longueur d'un chemin comme le nombre d'arcs qui le composent.

De nombreux problèmes se modélisent comme une recherche de plus court chemins entre deux points d'un graphe. Mais bien souvent les arcs n'ont pas tous la même longueur. Nous allons donc généraliser le parcours en largeur pour pouvoir calculer des pus courts chemins lorsque chaque arc possède une longueur positive ou nulle.

Nous posons donc le problème ainsi. Soit un graphe orienté $G = (V, E)$ et une fonction déterminant la longueur de chaque arc $l : E \to \mathbb{R}^+$. Nous redéfinissons alors la *longueur d'un chemin* par $l(e1, e_2, \ldots, e_k) = l(e_1) + l(e_2) + \ldots + l(e_k)$. Étant donné une source $s \in V$, déterminer pour tout sommet u accessible depuis s un chemin de longueur minimum.

Définition 2.7.1. *La* distance *entre deux sommets s et t d'un graphe $G = (V, E)$ pour une fonction de longueur $l : E \to \mathbb{R}^+$ est définie par* $\min_{P\ :\ st-chemin} l(P)$ *et est notée $d_l(s, t)$ (ou plus simplement $d(s, t)$ s'il n'y a pas d'ambiguïté).*

Nous avons choisi de prendre des longueurs positives réelles. Pourquoi ne pas les prendre négatives ? La présence de longueur négative complique énormément le problème, et n'est souvent pas naturelle. Cela oblige au mieux à utiliser des algorithmes plus complexes, et au pire, il n'existe peut-être même pas d'algorithme polynomial. Nous nous concentrons donc sur le cas des longueurs positives, à la fois plus simple et plus naturel. Quand au choix des réels, il s'avère que l'algorithme que nous allons voir nous permet cette généralité. Bien sûr programmer avec des réels est un exercice délicat, nous supposons ici que l'addition et la comparaison de deux réels sont des opérations élémentaires.

Nous sommes donc confrontés à ce problème de trouver les plus courts chemins depuis le sommet s de $G = (V, E)$. Nous allons procéder à une expérience de pensée afin de découvrir un algorithme. Imaginons que le graphe est un objet physique, les sommets étant de larges places, et les arcs des avenues à sens unique. Nous nous trouvons en s et cherchons un plus court chemin vers v. Une façon naïve de procéder serait de tester toutes les directions possibles depuis s, puis toutes les directions possibles depuis le second sommet, et ainsi de suite.

Comme le nombre d'arcs sortants d'un sommet peut être élevé, le nombre de choix croît exponentiellement, voir plus, à chaque sommet où se pose un choix. Un seul homme ne peut donc y parvenir. Mais là où un homme échoue, une foule peut réussir ! Faisons donc partir simultanément de s, marchant à la même vitesse, des groupes d'individus dans chaque direction. Lorsqu'un groupe atteint un sommet, il lui suffit de se diviser en suffisamment de groupes pour à nouveau partir dans toutes les directions. Il suffit pour cela que le groupe soit assez nombreux, et cela ne nous coûte rien dans cet expérience de pensée de supposer que ce sera toujours le cas. Ainsi tous les chemins possibles depuis s sont testés par au moins un groupe. Le premier groupe à atteindre le sommet v l'aura donc fait via un plus court sv-chemin.

Cela fait beaucoup de groupes à gérer, mais nous allons pouvoir en supprimer assez facilement. Si un groupe arrive à un sommet mais qu'il s'est fait précéder par un autre groupe, alors les membres de ce dernier ont pris une avance qui n'est pas rattrapable. Le groupe arrivé en retard peut donc abandonner sa quête : il serait toujours devancé dans un sommet où il arrive. Ainsi, dans chaque sommet seul le premier groupe se divise et repart par les arcs sortants. Donc le nombre de groupes nécessaires pour tout l'algorithme est en fait égal au nombre d'arcs accessibles depuis s.

Il nous faut maintenant gérer les départs et arrivées des groupes. Pour cela, remarquons que si nous connaissons la date de départ d'un groupe et la longueur de l'arc par lequel il part, nous pouvons par addition trouver la date d'arrivée. Nous allons donc avoir des événements "tel groupe arrive à tel sommet", chacun avec sa date. Nous pouvons donc traiter successivement ces événements par ordre chronologique. Chaque fois qu'un groupe arrive à un sommet, nous créons si nécessaire les nouveaux événements pour les groupes qui repartent de ce sommet s'il y en a. Et nous traitons les différents événements dans l'ordre dans lesquels ils se présentent.

Ce que nous venons de décrire est en fait un parcours de graphe. Créer un nouveau groupe, c'est simplement ajouter un arc (un événement en fait) dans la structure d'insertion-extraction. Résoudre un événement, c'est extraire l'événement de date minimum de cette structure, vérifier que le groupe arrive ou pas à un nouveau sommet, et si nécessaire faire partir de nouveaux groupes sur chaque arc sortant, autrement dit appeler la fonction explore. Nous avons donc besoin donc besoin d'une structure de files de priorité (un tas) sur les dates pour la structure d'insertion-extraction.

Lors de la découverte d'un nouveau sommet, nous notons la longueur du plus court chemin comme étant la date actuelle. Les groupes qui repartent ont pour date d'arrivée la date actuelle plus la longueur des arcs par lesquels ils partent. Nous pouvons donc préciser l'algorithme en Figure 2.26. Cet algorithme porte le nom de son premier découvreur, Edsger Dijkstra.

Nous prouvons la correction de cet algorithme. Nous commençons par démontrer que les événements sont traités dans l'ordre chronologique, comme prévu.

Lemme 2.7.1. *Les priorités des arcs extraits de* frontière *en ligne 15, Figure 2.26, forment une séquence croissante.*

```
1   fonction plus_courts_chemins(graphe g, réel longueur(arc), sommet s)
2       : (tableau d'indice sommet d'(arc ou bien ⊥), tableau d'indice sommet de réel)
3     soit predecesseur := tableau d'indice sommets(g) de arc ou bien ⊥
4     soit distance := tableau d'indice sommets(g) de réel
5     soit parcouru := {s}
6     soit frontière := Tas.vide()
7
8     soit fonction explore(sommet u) : void =
9       pour tout e ∈ arc_sortants(g, u) faire
10        Tas.insère(e, distance[u] + longueur(e), frontière)
11
12    predecesseur[s] ← ⊥, distance[s] ← 0
13    explore(g, s, frontière)
14    tant que ¬Tas.est_vide(frontière) faire
15      soit e := Tas.extrais(frontière); soit v := tete(e)
16      si v ∉ parcouru alors
17        parcouru ← parcouru ∪ {v}; predecesseur[v] ← e
18        distance[v] ← longueur(e) + distance[queue(e)]
19        explore(g, v, frontière)
20    retourner (predecesseur, distance)
```

FIGURE 2.26 – L'algorithme de Dijkstra (version simple), pour une structure de file de priorité minimum *TTas*.

Démonstration. Il suffit de remarquer qu'entre deux extractions, le minimum de la file de priorité ne peut qu'augmenter. Il augmente lors des opérations d'extraction évidemment, de plus lors de l'insertion d'un nouvel élément, ligne 10, cet élément est inséré avec une priorité égale à la somme de la priorité du minimum qui vient d'être extrait plus la longueur de l'élément. L'élément inséré a donc une priorité supérieure à l'élément extrait. □

Théorème 2.7.1. *L'algorithme de Dijkstra est correct. Sa complexité est* $O(|V(G)|+|E(G)|\log|V(G)|)$.

Démonstration. Nous devons démontrer que les distances et prédécesseur de chaque sommet sont bien calculés. Par l'absurde, supposons que ce n'est pas le cas. Soit v le sommet minimisant $d(l, v)$ pour lequel l'algorithme calcule de mauvaises valeurs. Clairement $v \neq [\![\mathsf{s}]\!]$.

Examinons l'itération de la boucle **tant que** ligne 14 pour laquelle $[\![\mathsf{v}]\!] = v$ pour la première fois. Par le choix de v, la distance $u = [\![\mathsf{queue(v)}]\!]$ a été correctement calculée, ainsi que le prédecesseur de u.

Soit $e_1, e_2, \ldots, e_k$ un sv-chemin de longueur $d(s, v)$. Alors $e_k \neq uv$, puisque sinon la distance de v aurait été correctement calculée. Donc $e_k = wu$ avec $w \neq u$. De plus par positivité des longueurs $d(s, w) \leq d(s, v) < [\![\mathsf{distance[v]}]\!]$, donc par le Lemme 2.7.1 w est déjà parcouru lors de cette itération. Donc l'arc wv fut inséré dans la frontière, mais pas encore retiré. Or sa priorité est plus petite que celle de e qui est choisi, ce qui contredit la spécification des files de priorités.

Sa complexité est une conséquence directe du Lemme 2.2.2. □

Il est possible de coder cet algorithme plus efficacement. La complexité asymptotique de cette version est dominée par les opérations sur la file de priorité. Pour la réduire, il faut deux choses :

— réduire la taille de la frontière,

— utiliser des opérations de files de priorités plus rapides.

Pour le premier point, nous pourrions faire en sorte de ne garder dans la structure qu'au plus un arc vers chaque sommet. En effet, seul l'arc de plus petite priorité peut produire un plus court chemin. Ceci implique que pour le second point, il nous faut une opération qui remplace un arc vers un sommet v par un arc avec une priorité plus faible vers v. Autrement dit, il faut une opération qui diminue la priorité d'une clé de la file de priorité. Les *tas de Fibonacci* sont une structure appropriée, permettant de faire décroitre la valeur d'une clé en temps presque constant.

2.8 Arbre couvrant de poids minimum

Exceptionnellement dans ce chapitre, nous considérerons des graphes non-orientés. Dans ce cas, on parle plutôt d'*arêtes* que d'arcs, qui sont des ensembles de deux sommets, c'est-à-dire qu'il n'y a pas de sommet distingué comme la tête et l'autre comme la queue, une arête n'a pas de sens. Nous pouvons faire cependant des parcours de graphe non-orienté de la même façon que dans les graphes orientés. La seule adaptation est de remplacer la notion d'arcs sortants d'un sommet par l'ensemble des arcs incidents au sommet. Ainsi chaque parcours utilisera chaque arête deux fois, une fois dans un sens et une fois dans l'autre.

Nous aurons aussi besoin dans cette section de quelques définitions et rappels :

Définition 2.8.1. *Soit $G = (V, E)$ un graphe non-orienté. Pour tout $X \subseteq V$, Notons $\delta(X) = \{uv \in E : u \in X, v \notin X\}$, la* coupe *de* bord *$X$.*

Proposition 2.8.1. *Pour un graphe G non-orienté,*

(i) la relation être accessible depuis *est une relation d'équivalence.*

(ii) u n'est pas accessible depuis v s'il existe $X \subset V$ avec $v \in X$, $u \notin X$ et $\delta(X) = \emptyset$. Nous disons alors que $\delta(X)$ sépare u de v.

(iii) G est connexe ssi il n'existe pas $X \subsetneq V$, $X \neq \emptyset$ tel que $\delta(X) = \emptyset$.

(iv) l'intersection d'un cycle et d'une coupe contient un nombre pair d'arcs.

Démonstration. À faire en exercice. □

Nous nous posons la question de trouver un ensemble d'arêtes qui suffisent à connecter entre eux tous les sommets d'un graphe connexe G.

Définition 2.8.2. *Un* sous-graphe *d'un graphe $G = (V, E)$ est un graphe $H = (V', F)$ avec $V' \subseteq V$, $F \subseteq E$ tel que pour tout $e = uv \in F$, $u \in V'$ et $v \in V'$. H est dit* couvrant *si $V' = V$. Un graphe est* connexe *si pour toute paire de sommets u, v, il existe un uv-chemin.*

Ici, la différence avec fortement connexe est que la notion de chemin est plus faible : les arêtes peuvent être utilisées dans n'importe quel sens.

Nous cherchons donc un sous-graphe couvrant de G, qui soit connexe. Un tel problème modélise par exemple la création d'un réseau entre des clients, les arêtes représentant des connexions élémentaires potentielles. Nous pouvons modéliser le coût pour établir une de ces connexions en attribuant à chaque arête une valeur $c : E \to \mathbb{R}^+$.

Le problème se résume donc ainsi :

Définition 2.8.3. *Le* problème de l'arbre couvrant de coût minimum *est défini ainsi : étant donné un graphe connexe $G = (V, E)$ non-orienté et $c : E \to \mathbb{R}^+$, trouver $F \subset E$ tel que (V, F) soit connexe et $c(F) = \sum_{e \in F} c(e)$ soit minimum.*

Il est facile de voir qu'une solution optimale ne peut pas comporter de cycle : on pourrait alors enlever un arc quelconque du cycle, ce qui maintiendrait le sous-graphe connexe mais d'un coût plus petit. Un graphe sans cycle s'appelle un arbre, ce qui justifie le nom du problème.

Il existe plusieurs algorithmes pour déterminer l'arbre de coût minimum. Nous étudions ici l'algorithme de Prim qui utilise un parcours du graphe. Rappelons que les parcours construisent des arborescences (un arbre dans les graphes non-orientés) vers la source, grâce à la fonction prédecesseur. Notre approche est de spécialiser le parcours pour que l'arborescence soit un arbre couvrant de poids minimum.

L'algorithme de Prim consiste à utiliser une file de priorité comme structure d'insertion-extraction, tout comme dans l'algorithme de Dijkstra, mais avec des priorités différentes. Dans l'algorithme de Prim, la priorité d'une arête est simplement son poids. L'arbre construit est celui des arcs se trouvant dans le tableau prédecesseur à la fin de l'algorithme.

Nous démontrons la correction de cet algorithme, en commençant par un lemme général sur les arbres couvrants de coût minimum.

Lemme 2.8.1. *[de l'arête minimale] Soit $T = (V, F)$ un arbre couvrant minimal de $G = (V, E)$, $c : E \to \mathbb{R}^+$, et $X \subsetneq V$, $X \neq$. Alors il existe $e_X \in \delta(X) \cap F$ tel que $c(e_X) = \min_{e \in X} c(e)$ (un arc de plus petit coût de chaque coupe est contenu dans F).*

Démonstration. Soit $\delta(X)$ une coupe et $e = uv \in \delta(X) \setminus F$. Alors $F \cup \{e\}$ contient un cycle C, et $C \cap \delta(X)$ est pair donc $F \cap \delta(X)$ contient une arête f. Clairement $(V, F \cup \{e\} \setminus \{f\})$ est connexe, donc $c(f) \leq c(e)$. $F \cap \delta(X)$ contient un arc de poids inférieur à tout autre arc de $\delta(X)$, il contient donc un arc de coût minimum de $\delta(X)$. □

Théorème 2.8.1. *L'algorithme de Prim pour le calcul d'un arbre couvrant de poids minimum est correct et sa complexité est $O(|V(G)| + |E(G)| \log |V(G)|)$.*

Démonstration. Supposons d'abord que chaque arc a un coût distinct. Nous montrons que toute arête choisie par l'algorithme est l'unique arête minimale d'une coupe bien choisie.

Considérons l'arête $e = [\![\mathsf{e}]\!]$ en ligne 15 de l'algorithme de parcours, Figure 2.3. Cette arête est ajouté à prédecesseur, on doit donc trouver une coupe dont elle est l'arête de coût minimum. Soit $R := [\![\mathsf{parcouru}]\!]$ pris juste avant l'exécution de la ligne 15 et $F = [\![\mathsf{frontière}]\!]$ pris juste avant l'extraction en ligne 13. Alors $\delta(R) \subseteq F$ car la frontière contient en début d'itération tous les arcs depuis un sommet parcouru vers un sommet non-parcouru. Par propriété des files de priorité, e est l'arc de coût minimal dans F et donc dans $\delta(R) \subseteq F$. Par le Lemme 2.8.1, toutes les arêtes choisies sont obligatoirement dans l'arbre de coût minimum, donc l'algorithme de Prim est correct si tous les coûts sont distincts.

Si les coûts des arêtes ne sont pas distincts, il suffit de les modifier : soit c' la fonction de coût $c' : e \to c(e) + \epsilon_e$ avec $0 \leq \epsilon_e < \epsilon/|E|$ des valeurs distinctes pour chaque arête, et $\epsilon = \min\{|c(A) - c(B)| \ : \ A, B \subseteq E, c(A) \neq c(B)\}$. Alors tous les arcs ont des valeurs différentes :

$$c'(e) = c'(f) \implies \epsilon \geq |c(e) - c(f)| = |\epsilon_e - \epsilon_f| < \epsilon$$

contradiction. De plus, l'arbre de coût minimum pour c' est un arbre de coût minimum pour c :

$$\begin{aligned} & c'(F) \geq c'(F') \\ \implies & c(F) - c(F') \geq \sum_{e \in F} \epsilon_e - \sum_{e \in F'} \epsilon_e \\ \implies & c(F) - c(F') > -\epsilon \\ \implies & c(F) \geq c(F') \end{aligned}$$

Enfin, en choisissant $\epsilon_e < \epsilon_{e'}$ si e est extrait avant e' dans l'algorithme de Prim pour c, les arcs sont extraits dans le même ordre pour c et pour c' : comme l'algorithme pour c' est correct, il l'est aussi pour c.

La complexité est de nouveau une conséquence immédiate du Lemme 2.2.2. □

L'argument consistant à changer très légèrement les coûts des arêtes pour obtenir l'unicité des coûts est une technique générale appelée *perturbation*.

Chapitre 3

Algorithmes récursifs : programmation dynamique et diviser-pour-regner

3.1 Définition par récursion

Une définition par récurrence d'une fonction est une définition qui utilise la fonction elle-même pour se définir. Bien sûr, pour que la définition soit bien posée, certaines règles doivent être respectées. Pour qu'une définition par récurrence d'une fonction $f : A \to B$ soit correcte, nous imposons trois règles sur le graphe G_f dont l'ensemble des sommets est A, et il existe un arc $a \to b$ si la définition de $f(a)$ se réfère à $f(b)$. Les règles sont :

complétude : pour tout $a \in A$, $f(a)$ est défini par une expression faisant un nombre d'appels à f fini (éventuellement 0).

acyclicité : G_f est acyclique.

fondement : G_f ne possède pas de chemin de longueur infinie.

Le traditionnel oubli du cas de base illustre le manquement à la règle de complétude : il faut bien définir tous les cas, même ceux ne faisant pas intervenir d'appels récursifs. L'acyclicité et le fondement assurent que l'évaluation de f termine, et donc que f est bien définie.

Prenons par exemple la fonction $f : \mathbb{R} \to \mathbb{N}$ sur les réels définie par $f(x) = 1 + f(x/2)$ si $x > 0$, $f(x) = 0$ sinon. Il s'agit presque de la définition du logarithme en base 2, sauf que cette définition est erronée. Les propriétés de complétude et d'acyclicité sont vérifiées : G_f a pour sommets les réels, et l'ordre naturel sur les réels est un ordre topologique de G_f, la récursion de x vers $x/2$ respecte cet ordre puisque si $x > 0$, $x > x/2$. Par contre, la règle de fondement n'est pas correcte : il existe un chemin infini, par exemple $1 \to 1/2 \to 1/4 \to 1/8 \to \cdots$. La définition correcte est $f(x) = 1 + f(x/2)$ si $x \geq 1$, $f(x) = 0$ sinon. Alors, si $x \geq 1$, x décroit d'au moins $x/2 \geq 1/2$, ce qui nous assure que le chemin partant de x a une longueur d'au plus $2x$ (en fait beaucoup moins en général).

Dans le cas fréquent où $A = \mathbb{N}$ et la définition de $f(n)$ dépend de certaines valeurs $f(p)$ avec $p < n$, le graphe G_f que nous utilisons est un sous-graphe du graphe de l'ordre total naturel, défini par $n \to p$ si $n > p$. Les chemins dans ce graphe définissent donc des séquences strictement décroissantes et positive, donc finie.

Nous allons voir des exemples pour lesquels A est un ensemble plus compliqué : l'ensemble des arbres, des matrices, des mots... Comme dans le cas des entiers, le plus souvent G_f est un sous-graphe d'un graphe d'ordre bien-fondé.

Définition 3.1.1. *Un ordre est dit* bien-fondé *s'il n'existe pas de suite $(e_i)_{i\in\mathbb{N}}$ strictement décroissante $e_0 > e_1 > e_2 > \ldots$ (ce qui implique la propriété de fondement de son graphe).*

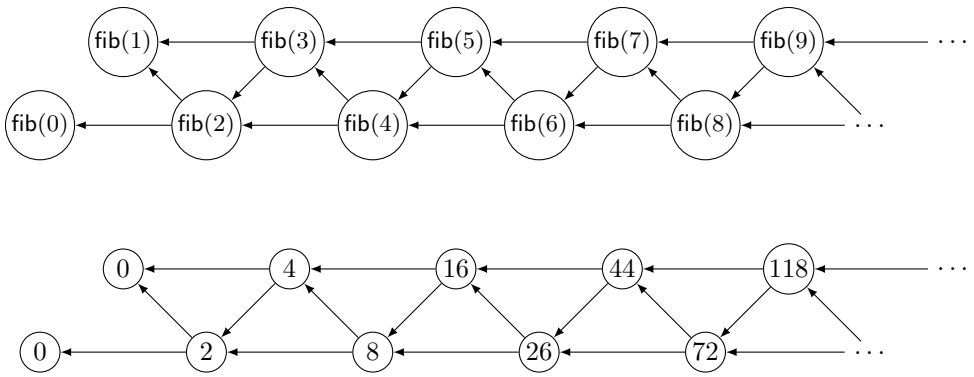


FIGURE 3.1 – Une représentation d'une partie du graphe G_{fib} en haut. En bas, le nombre d'appels récursifs nécessaires avec la version naïve pour le calcul d'un terme de la suite de Fibonacci.

3.2 Calcul ascendant d'une fonction récursive (ou programmation dynamique)

Une façon naïve de calculer une fonction récursive est la suivante. Pour évaluer $f(x)$, nous évaluons indépendamment tous les termes $f(x_0), f(x_1), \ldots, f(x_k)$ apparaissant dans la définition de $f(x)$, avant d'évaluer $f(x)$ lui-même.

Pour comprendre pourquoi cette méthode est naïve, il suffit de reprendre l'exemple classique de la suite de Fibonacci, définie par

$$\begin{aligned} \mathsf{fib}(0) &= 1 \\ \mathsf{fib}(1) &= 1 \\ \mathsf{fib}(n+2) &= \mathsf{fib}(n+1) + \mathsf{fib}(n) \qquad \text{pour tout } n \geq 0 \end{aligned}$$

Avec la version naïve, chaque appel récursif sur un entier $n > 1$ provoque deux autres appels récursifs, qui eux-même en provoquent quatre autres, et ainsi de suite, le nombre d'appels récursifs augmentant de façon exponentielle, comme illustré par la Figure 3.1.

Une première façon de contourner ce problème (la façon traditionnelle), c'est de calculer les fonctions récursives de façon ascendante et non pas descendante. En version descendante, le calcul de $\mathsf{fib}(9)$ appelle le calcul de $\mathsf{fib}(8)$ et $\mathsf{fib}(7)$, etc. En version ascendante, nous commençons par calculer $\mathsf{fib}(0)$, puis $\mathsf{fib}(1)$, $\mathsf{fib}(2)$, $\mathsf{fib}(3)$, jusqu'à arriver à $\mathsf{fib}(9)$. À ce moment là, les valeurs de $\mathsf{fib}(8)$ et $\mathsf{fib}(7)$ étant connues (puisque 7 et 8 sont plus petits que 9), le calcul de $\mathsf{fib}(9)$ nécessite une simple addition, pour peu que nous ayons pris le soin de stocker (par exemple dans un tableau) les valeurs des calculs précédents.

Plus généralement pour un graphe G_f quelconque et un sommet v dont nous voulons calculer l'image, le calcul suit les étapes suivantes :

1. Calculer l'ensemble S des sommets accessibles depuis v,
2. Ordonner ces sommets S selon un tri topologique ($u \to v$ implique $v < u$),
3. Par ordre croissant, calculer et stocker l'image de chaque sommet de S.

Souvent l'une ou l'autre de ces étapes est triviale, comme dans le cas de Fibonacci : l'ensemble des entiers accessibles depuis n est $[0, n]$, et l'ordre est l'ordre naturel sur les entiers. C'est cette technique qui est appelé *programmation dynamique*.

Dans ce cas, la complexité du calcul est dominé par le calcul de chaque sommet de S. Pour Fibonacci, chaque sommet nécessite au plus une addition, et pour $\mathsf{fib}(n)$, il y a $n+1$ sommets accessibles, donc cela donne une complexité de $\sum_{i=0}^{n} O(1) = O(n)$.

Une façon un peu plus élaborée de calculer une telle fonction récursive est une technique de programmation appelée *mémoïsation*. Il suffit de remarquer que la seule étape vraiment utile dans la

```
1  soit fib_table := [1, 1, ⊥, ⊥, ...] : tableau d'entier
2
3  fonction fib(entier n) : entier
4    si fib_table[n] = ⊥ alors
5      fib_table[n] ← fib(n − 1) + fib(n − 2)
6    retourner fib_table[n]
```

FIGURE 3.2 – Fibonacci mémoïsé

technique de programmation dynamique, c'est celle consistant à stocker les valeurs intermédiaires. Ainsi l'algorithme mémoïsé teste dans un premier temps si la valeur d'entrée de la fonction a déjà été calculée. Si c'est le cas, il suffit d'accéder au résultat stocké en mémoire. Sinon, l'algorithme effectue le calcul normalement, avec d'éventuels appels récursifs à l'algorithme mémoïsé, puis stocke le résultat en mémoire. Dans le cas de Fibonacci, l'algorithme mémoisé est donné en Figure 3.2 (il faut choisir un tableau d'entiers suffisamment grand pour stocker toutes les valeurs à considérer).

La mémoïsation rend inutile le calcul de l'ordre dans lequel évaluer les sommets. En échange, il faut pouvoir stocker facilement les valeurs calculées, ce qui nécessite une structure de données appropriée, selon les arguments de l'algorithme. Souvent des tableaux suffisent, sinon le recours à une structure de dictionnaire, par exemple les tables de dispersion (cf Section **??** du Chapitre **??**), est nécessaire.

Enfin, programmation dynamique ou mémoïsation ne sont pas toujours nécessaires, loin s'en faut. Ce sont deux techniques qui permettent de résoudre un problème de complexité, lorsque les appels récursifs se font sur le même petit ensemble de valeurs (l'ensemble S de sommets accessible dans G_f doit être petit) mais avec beaucoup de partage (le même sommet est appelé par de nombreux autres sommets de S). Dans beaucoup de fonctions récursives, chaque valeur est appelée au plus une fois en argument lors d'un calcul (par exemple, la factorielle, ou bien les exemples de la section suivante), alors la programmation dynamique n'apporte pas d'amélioration significative à la résolution du problème.

Comment reconnaitre un problème pouvant avoir un algorithme de programmation dynamique efficace ? La programmation dynamique a beaucoup d'applications très différentes, mais on retrouve bien souvent un même schéma. Le principe est d'exprimer une valeur associée à un objet (la fonction que nous souhaitons calculer) par rapport aux valeurs associées à des objets de la même nature, mais plus petits. La question à se poser est donc : existe-t-il une famille d'objets plus petits, pour lesquels nous pourrions avoir une relation facile à exprimer sur la fonction à calculer ? De plus, pour être efficace, cette famille doit être petite.

Très souvent l'objet principal prend la forme d'une séquence $\langle e_1, e_2, \ldots, e_n \rangle$. Voici quelques familles qu'il est naturel de considérer :

- les préfixes de la séquence : $\langle e_1 \rangle$, $\langle e_1, e_2 \rangle$, $\ldots$, $\langle e_1, \ldots, e_{n-1} \rangle$,
- symétriquement les suffixes de la séquence : $\langle e_n \rangle$, $\langle e_{n-1}, e_n \rangle$, $\ldots$, $\langle e_2, \ldots, e_n \rangle$,
- les sous-séquences d'éléments consécutifs (chaînes), de la forme $\langle e_i, e_{i+1}, \ldots, e_{j-1}, e_j \rangle$ avec $i < j$,
- plus rarement, les paires de chaînes disjointes : $(\langle e_i, e_{i+1}, \ldots, e_j \rangle, \langle e_k, e_{k+1}, \ldots, e_l \rangle)$ avec $i < j < k < l$.

Dans les deux premiers cas, la famille est de taille n, dans le troisième cas de taille $O(n^2)$ et dans le dernier cas de taille $O(n^4)$.

Parfois il s'agit de plusieurs séquences, prenons le cas de deux séquences. Nous pouvons alors choisir le produit cartésien de deux familles, une pour chaque séquence. Par exemple :

- les paires de préfixes de chaque séquence, $O(n^2)$ éléments,
- les paires d'un préfixe de la première séquence et d'un suffixe de la seconde séquence, aussi $O(n^2)$ éléments,
- les paires d'un préfixe de la première séquence et d'une chaîne de la deuxième séquence, $O(n^3)$ éléments,
- etc.

Une matrice peut être vue comme une famille d'éléments indicés par deux séquences, ce qui permet d'appliquer les cas ci-dessus. Les familles les plus naturelles sont :

- les ensembles consécutifs de lignes (ou de colonnes) de la matrice, $O(n^2)$ éléments,
- les sous-matrices consécutives : l'intersection de lignes consécutives et de colonnes consécutives, $O(n^4)$ éléments,
- les sous-matrices consécutives contenant le coin supérieur gauche (ou tout autre coin), $O(n^2)$ éléments.

Il est d'ailleurs assez courant de représenter les couples de séquences avec la matrice du produit cartésien.

Anecdotiquement, le problème peut porter sur une structure mathématique qui ne correspond pas à un produit cartésien de séquences. Dans ce cas, trouver la bonne famille pour exprimer la récurrence est un pré-requis parfois complexe.

3.2.1 Exemple : plus longue sous-séquence commune

Définition 3.2.1. *Soit $S = \langle e_1, e_2, \ldots, e_n\rangle$ une séquence d'éléments d'un alphabet Σ. Une* sous-séquence *est une séquence $R \in \Sigma^\star$ de la forme $R = \langle e_{i_1}, e_{i_2}, \ldots, e_{i_l}\rangle$ avec $1 \leq i_1 < i_2 < \cdots < i_l \leq n$. On note $R \triangleleft S$ si R est une sous-séquence de S.*

Par exemple, pour l'alphabet $\Sigma = \mathbb{N}$, $\langle 3, 5, 4\rangle$ est une sous-séquence de $S = \langle 1, 3, 2, 6, 5, 4, 7\rangle$, mais pas $\langle 2, 3, 5\rangle$ car 2 doit apparaître après 3. Une sous-séquence s'obtient donc en supprimant des éléments mais sans changer l'ordre des éléments entre eux.

Étant données deux séquences S et T, nous voulons calculer la plus longue sous-séquence R apparaissant dans S et dans T.

Définition 3.2.2. *Une* plus longue sous-séquence commune *à deux séquences $S \in \Sigma^\star$ et $T \in \Sigma^\star$ est un élément de longueur maximum parmi :*

$$ssc(S, T) := \{R \in \Sigma^\star \mid R \triangleleft S \wedge R \triangleleft T\}$$

Nous notons $plssc(S, T)$ l'ensemble des plus longues sous-séquences communes à S et T.

Le nombre de sous-séquences possibles de S est $2^{|S|}$, puisque nous avons pour chaque élément le choix de le prendre ou de le laisser dans la sous-séquence. Il n'est donc pas raisonnable d'essayer toutes les sous-séquences de S pour trouver la plus longue sous-séquence commune.

Nous nous tournons donc vers la programmation dynamique. S et T sont des séquences, nous avons donc plusieurs possibilités pour essayer de trouver une relation de récurrence. Pour ce problème, nous allons regarder les suffixes de S et T. En effet, il semble naturel que la plus longue sous-séquence commune de S et T sera *presque* une plus longue sous-séquence commune aux séquences S' et T', obtenues en enlevant respectivement la première lettre de S et de T. Nous allons formaliser cette intuition.

Observons d'abord les propriétés suivantes de $\triangleleft$. Notons $a.\langle e_1, \ldots, e_l\rangle$ la séquence $\langle a, e_1, \ldots, e_l\rangle$, et pour un ensemble $L \subseteq \Sigma^\star$, notons $a.L$ l'ensemble $\{a.S \mid S \in L\}$.

Proposition 3.2.1. *Pour tous $a, b \in \Sigma$ et $S, T \in \Sigma^\star$ avec $a \neq b$:*

$$S \triangleleft T \implies a.S \triangleleft a.T \tag{3.1}$$
$$S \triangleleft T \implies S \triangleleft a.T \tag{3.2}$$
$$a.S \triangleleft a.T \implies S \triangleleft T \tag{3.3}$$
$$b.S \triangleleft a.T \implies b.S \triangleleft T \tag{3.4}$$

Démonstration. Laissée en exercice. □

Nous pouvons maintenant prouver un résultat sur les sous-séquences communes qui va nous donner immédiatement la solution.

Lemme 3.2.1. *Pour tous $a, b \in \Sigma$ et $S, T \in \Sigma^\star$ avec $a \neq b$,*

$$ssc(a.S, a.T) = a.ssc(S, T) \cup scc(S, T) \tag{3.5}$$
$$ssc(a.S, b.T) = ssc(a.S, T) \cup ssc(S, b.T) \tag{3.6}$$

Démonstration.

$$\begin{aligned}
\mathrm{ssc}(a.S, a.T) &= \{R \mid R \lhd a.S, R \lhd a.T\} \\
&\quad \text{(Définition)} \\
&\subseteq \{a.R' \mid a.R' \lhd a.S, a.R' \lhd a.T\} \cup \{b.R' \mid b \neq a, b.R' \lhd a.S, b.R' \lhd a.T\} \cup \{\varepsilon\} \\
&\quad \text{(Par distinction de la 1re lettre)} \\
&\subseteq a.\{R' \mid R' \lhd S, R' \lhd T\} \cup \{R \mid R \lhd a.S, R \lhd a.T\} \\
&\quad \text{(Équations (3.3) et (3.4))} \\
&= a.\mathrm{scc}(S, T) \cup \mathrm{scc}(S, T) \\
&\quad \text{(Définition)} \\
&\subseteq \mathrm{scc}(a.S, a.T) \\
&\quad \text{(Équation (3.2))}
\end{aligned}$$

Donc (3.5) est vraie.

$$\begin{aligned}
\mathrm{ssc}(a.S, b.T) &= \{R \mid R \lhd a.S, R \lhd b.T\} \\
&\quad \text{(Définition)} \\
&= \{a.R' \mid a.R' \lhd a.S, a.R' \lhd b.T\} \\
&\quad \cup \{b.R' \mid b.R' \lhd a.S, b.R' \lhd b.T\} \\
&\quad \cup \{c.R' \mid c \notin \{a, b\}, c.R' \lhd a.S, c.R' \lhd b.T\} \cup \{\varepsilon\} \\
&\quad \text{(Par distinction de la 1re lettre de } R) \\
&\subseteq \{R \mid R \lhd S, R \lhd b.T\} \cup \{R \mid R \lhd a.S, R \lhd T\} \cup \{R \mid R \lhd S, R \lhd T\} \\
&\quad \text{(Multiples applications de l'équation (3.4))} \\
&= \mathrm{scc}(S, b.T) \cup \mathrm{scc}(a.S, T) \cup \mathrm{scc}(S.T) \\
&\quad \text{(Définition)} \\
&= \mathrm{ssc}(S, b.T) \cup \mathrm{scc}(a.S, T) \\
&\quad \text{(Car } \mathrm{scc}(S, T) \subseteq \mathrm{scc}(S, b.T) \text{ par l'équation (3.2))} \\
&\subseteq \mathrm{scc}(a.S, b.T) \\
&\quad \text{(Équation (3.2))}
\end{aligned}$$

Donc (3.6) est vraie. □

Ceci nous donne immédiatement la formule de récurrence :

Corollaire 3.2.1. *Pour tous $a, b \in \Sigma$, $S, T \in \Sigma^\star$ avec $a \neq b$:*

$$plscc(\varepsilon, S) = \{\varepsilon\} \tag{3.7}$$
$$plssc(a.S, a.T) = a.plssc(S, T) \tag{3.8}$$
$$plssc(a.S, b.T) = maximaux(plssc(S, b.T) \cup plssc(a.S, T)) \tag{3.9}$$

Ces formules de récurrences sont clairement basées sur les suffixes des séquences initiales. Pour mémoïser les valeurs des sous-problèmes pour une paire (S, T), de la forme (S', T') avec S' et T' suffixes de S et T respectivement, nous utilisons un tableau à deux dimensions, tel que

```
1   fonction plssc(tableau d'entier s, tableau d'entier t) : tableau d'entiers =
2     soit solution := tableau[0..longueur(s)][0..longueur(t)] de listes d'entiers
3     pour tout i ∈ ⟦0, longueur(s)⟧ solution[i][0] ← ⟨⟩
4     pour tout j ∈ ⟦0, longueur(t)⟧ solution[0][j] ← ⟨⟩
5     pour tout i ∈ ⟦1, longueur(s)⟧ faire
6       pour tout j ∈ ⟦1, longueur(t)⟧ faire
7         solution[i][j] ←
8           si s[i] = t[j] alors insère(s[i], solution[i − 1][j − 1])
9           sinon si longueur(solution[i][j − 1]) > longueur(solution[i − 1][j]) alors solution[i][j − 1]
10          sinon solution[i − 1][j]
11    retourner convertis_en_tableau(solution[longueur(s)][longueur(t)])
```

FIGURE 3.3 – Algorithme de recherche d'une plus longue sous-séquence commune, par calcul ascendant.

```
1   fonction plus_longue_sous_séquence_commune(liste d'entier s, liste d'entier t) : liste d'entier =
2     soit solution := tableau[longueur(s)][longueur(t) de liste d'entier ou ⊥
3
4     soit fonction plssc(liste d'entier s, liste d'entier t) : liste d'entier =
5       soit ls := longueur(s); soit lt := longueur(t)
6       si solution[ls][lt] = ⊥alors
7         solution[ls][lt] ←
8           si ls = 0 ∨ lt = 0 alors liste_vide
9           sinon si tête(s) = tête(t) alors insère(tête(s), plssc[ls − 1][lt − 1])
10          sinon si longueur(plssc(ls − 1, lt)) > longueur(plssc(ls, lt − 1) alors plssc[ls − 1][lt]
11          sinon plssc[ls][lt − 1]
12      retourner solution[ls][lt]
13
14    retourner plssc(longueur(s), longueur(t))
```

FIGURE 3.4 – Algorithme de recherche d'une plus longue sous-séquence commune, avec mémoïsation.

l'élément d'indice i, j représente la paire (S', T') avec $|S'| = i$ et $|T'| = j$. Le code est alors donné en Figure 3.3 avec un calcul ascendant de la formule récursive, et en Figure 3.4 pour une version mémoïsée. Notez qu'il est plus naturel d'utiliser des listes que des tableaux pour la version mémoïsée puisqu'elle utilise directement la récursion, et que les listes sont particulièrement adaptées aux algorithmes récursifs.

Lemme 3.2.2. *Les algorithmes des Figures 3.3 et 3.4 ont une complexité asymptotique de $O(nm)$ où n et m sont les longueurs respectives des deux séquences.*

Démonstration. Le résultat est immédiat pour le premier algorithme, Figure 3.3 : sa complexité est dominée par l'exécution de la double boucle, donc chacune des nm itérations prend un temps $O(1)$, en supposant que la structure de liste utilisée supporte le calcul de la longueur en temps constant.

Pour le deuxième algorithme, Figure 3.4, il faut compter le nombre d'appels à la fonction interne plssc. Celle-ci est appelé une fois ligne 14, plus 5 fois aux lignes 8 à 11. Ces dernières ne sont appelées qu'une seule fois par case du tableau solution lequel comporte $(n+1)(m+1)$ cases. Cela somme donc à $O(nm)$ appels à plssc. Sans compter les appels récursifs un appel élémentaire à plssc prend un temps constant, donc l'algorithme a pour complexité asymptotique $O(nm)$. □

En fait la version mémoïsée peut-être plus rapide que la version ascendante : cette dernière calcule toujours tous les sous-problèmes, mais la version mémoïsée peut en ignorer. Dans le cas extrême, s = t et plssc n'est appelé que sur les suffixes de même longueur, soit une complexité de

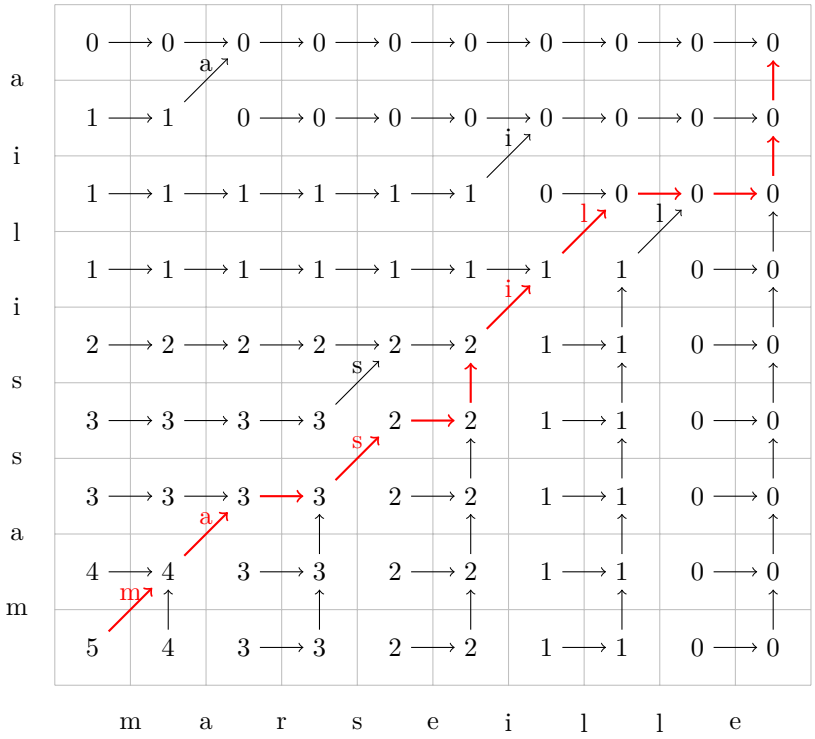


FIGURE 3.5 – Représentation par tableau du calcul de la plus longue sous-séquence commune des mots `marseille` et `massilia`. Chaque case correspond au sous-problème sur les suffixes formés respectivement avec les lettres des colonnes à droite et avec les lettres des lignes au-dessus. La valeur dans la case est la longueur d'une sous-séquence maximale. Les flèches indiquent quel est le maximum choisi pour trouver une case. La plus longue sous-séquence trouvée est donc `masil`.

$\Theta(n)$ dans ce cas. Ceci conduit à écrire des algorithmes plus sophistiqués pour explorer l'arbre des récursions, dans un ordre permettant de trouver assez vite de très bonnes solutions et d'éviter de résoudre des sous-problèmes qui n'ont aucune chance d'intervenir dans une solution optimale.

3.2.2 Structure secondaire d'une séquence d'ARN

L'ARN est un des composants fondamentaux de la biologie cellulaire. Tout comme l'ADN, qui code l'information génétique, l'ARN est une molécule constituée de quatres bases organisées en séquences. L'ARN comporte un seul brin, là où l'ADN en comporte deux organisés en double hélice. Un brin d'ARN est une séquence des bases adénine (A), cytosine (C), guanine (G) et uracil (U), que nous représentons par une séquence sur l'alphabet $\Sigma = \{A, C, G, U\}$.

Les propriétés d'une molécule d'ARN sont déterminées en partie par l'agencement spatial du brin. Celui-ci tend à faire des boucles, de façon à créer des liens supplémentaires entre des paires de bases non-consécutives *A*-*U* ou *C*-*G*. Décrire ces appariements, c'est donner ce qu'on appelle la structure secondaire de l'ARN. La Figure 3.6 présente un exemple de cette structure, qui montre comment les appariements influencent les repliements de la molécule.

Définition 3.2.3. *Notons $w_1 w_2 \cdots w_l \in \Sigma^l$ une séquence d'ARN. Une* structure secondaire *est un ensemble $S \subset C_l^2$ de paires d'indices vérifiant les propriétés :*
(i) pour tout $\{i, j\} \in S$, $\{w_i, w_j\} \in \{\{A, U\}, \{C, G\}\}$,
(ii) pour tout $\{i, j\} \in S$, $|i - j| > 4$,
(iii) il n'existe pas $\{i, j\}, \{k, l\} \in S$ avec $\{i, j\} \cap \{k, l\} \neq \emptyset$,
(iv) il n'existe pas $\{i, j\}, \{k, l\} \in S$ avec $i < k < j < l$.

La propriété (i) stipule que seules des paires de bases complémentaires peuvent se former. La propriété (ii) interdit que des paires de bases trop proches se forment : il s'agit d'une contrainte sur la souplesse du brin d'ARN, qui ne peut pas être plié trop fort. La contrainte (iii) exprime que chaque base apparait dans au plus une paire. La dernière contrainte (iv), la contrainte de décroisement, indique que deux paires ne peuvent pas *se croiser*. Ces contraintes sont plus facilement visibles en *étalant* le brin d'ARN sur une droite et en représentant les appariements par des lignes courbes, qui ne se croisent pas. L'exemple de la Figure 3.6 est ainsi repris en Figure 3.7.

Les règles concernant la formation de cette structure secondaire sont complexes, mais sont souvent bien approximées par l'appariement maximum : celui contenant le plus de paires. La question qui nous intéresse est donc de trouver un appariement maximum.

Ici, la structure même de la solution est récursive : si on se restreint à une *chaîne*, c'est-à-dire une sous-séquence de bases consécutives, les appariements entre bases de cette sous-séquence doivent aussi vérifier les quatre conditions. Cela nous incite à résoudre les sous-problèmes définis par les chaînes, et recombiner ces solutions pour obtenir l'appariement maximum.

Il nous faut pour cela obtenir une formule de récurrence sur la taille de l'appariement maximum pour un mot quelconque.

Définition 3.2.4. *Pour tout mot $w \in \Sigma^\star$, notons $\nu(w)$ le nombre maximum de paires d'un appariement de w.*

Proposition 3.2.2. *Soient $u, v \in \Sigma^\star$, alors $\nu(u.v) \geq \nu(u) + \nu(v)$. Si de plus $\{x, y\}$ est une paire de base compatible et $|u| \geq 4$, alors $\nu(x.u.y.v) \geq 1 + \nu(u) + \nu(v)$.*

Démonstration. L'union $S_{u.v}$ des appariements maximums S_u de u et S_v de v est un appariement de $u.v$, avec

$$S_{u.v} := \{\{i, j\} \mid \{i, j\} \in S_u\} \cup \{\{i + |u|, j + |u|\} \mid \{i, j\} \in S_v\}$$

les propriétés (i) à (iv) étant clairement vérifiées pur $S_{u.v}$.

La deuxième partie est similaire. □

Lemme 3.2.3. *Pour tout mot $w = w_1 \cdots w_l \in \Sigma^\star$ avec $l > 4$, et S un appariement maximal pour w :*

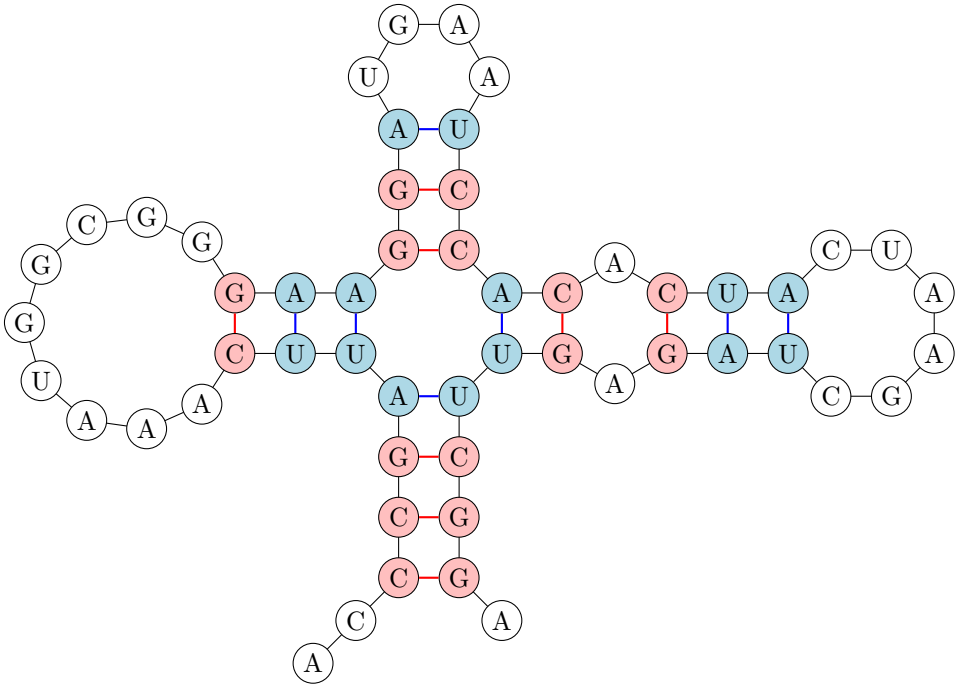


FIGURE 3.6 – Un exemple de structure secondaire d'un brin d'ARN.

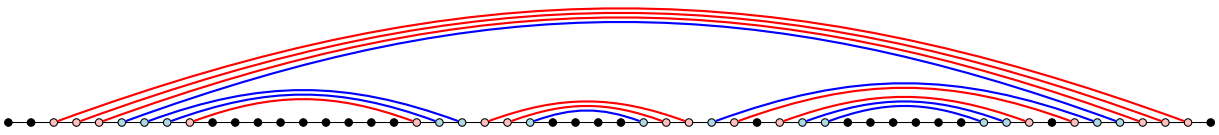


FIGURE 3.7 – La même structure secondaire présentée *à plat*.

```
1   type base = A ou bien C ou bien G ou bien U
2
3   fonction compatible(base x, base y) : booléen =
4     retourner{x, y} = {A, U} ∨ {x, y} = {C, G}
5
6      // On suppose que opt[i][j] = ⟨⟩ si i > j
7   fonction appariement(tableau de base arn) : liste de (entier, entier) =
8     soit n = longueur(arn)
9     soit opt = tableau[1..n][1..n] d'ensemble de (entier, entier)
10    pour début de 1 à n par pas de −1 faire
11      pour fin de début à n faire
12        opt[début][fin] ←
13            si fin − début < 4 alors ⟨⟩
14            sinon opt[début + 4][fin]
15        pour milieu de début + 1 à fin faire
16          si compatible(arn[début], arn[fin])
17            ∧|opt[début + 1][milieu − 1]| + |opt[milieu + 1][fin]| > |opt[début][fin]|
18          alors opt[début][fin] ← {(début, fin)} ∪ opt[début + 1][milieu − 1] ∪ opt[milieu + 1][fin]
19    retourner opt[1][n]
```

FIGURE 3.8 – Algorithme d'appariement d'une séquence d'ARN

— *soit il existe* $j \in [\![2, l]\!]$ *tel que* $\{1, j\} \in S$*. Dans ce cas,* $j \geq 5$ *et* $\nu(w) = 1 + \nu(w_2 \cdots w_{j-1}) + \nu(w_{j+1} \cdots w_l)$.
— *sinon,* $\nu(w) = \nu(w_2 \cdots w_l)$.

Démonstration. Premier cas, il existe j avec $\{1, j\} \in S$. Par la propriété (ii), $j > 4$. Notons $u := w_2 \cdots w_{j-1}$ et $v := w_{j+1} \cdots w_l$, donc $w = w_1.u.w_j.v$. Soit S_u l'ensemble des paires $\{\{i, k\} \in S \mid i < j, k < j\}$ et S_v l'ensemble des paires $\{\{i, k\} \mid i > j, k > j\}$. Alors $\{S_u, S_v\}$ partitionne $S \setminus \{\{1, j\}\}$ par la propriété (iv) et l'existence de la paire $\{1, j\}$ dans S. De plus S_u et S_v sont des appariements de u et v respectivement. Donc par la proposition précédente, $\nu(w) = 1 + |S_u| + |S_v| \leq 1 + \nu(u) + \nu(v) \leq \nu(w)$.

Deuxième cas, comme w_1 n'est pas apparié dans l'appariement maximum S, S induit un appariement sur $w_2 \cdots w_l$, donc $\nu(w_2 \cdots w_l) \geq \nu(w) \geq \nu(w_2 \cdots w_l)$. □

Le Lemme 3.2.3 nous fournit une formule de récurrence pour le calcul de l'appariement maximum. Il faut lui ajouter comme cas de base $\nu(w) = 0$ si $|w| \leq 4$. Les sous-problèmes utilisés par cette formule sont effectivement des chaînes de la séquence initiale, et le nombre de ces chaînes est quadratique dans la longueur de la séquence. Le calcul de l'optimum pour une chaîne prend un temps linéaire (en supposant que le cardinal et l'union d'un ensemble est calculé en $O(1)$, ce qui est le cas en utilisant des listes avec un champ pour mémoriser leurs longueurs) : c'est le nombre de cas à tester selon le Lemme 3.2.3 pour le choix de j. Nous en déduisons la complexité de l'algorithme obtenu, présenté en Figure 3.8.

Lemme 3.2.4. *La complexité de l'algorithme de la Figure 3.8 pour le calcul d'un appariement maximum d'une séquence d'ARN de longueur* n *est* $O(n^3)$ *asymptotiquement dans le pire des cas.*

L'agorithme décrit en Figure 3.8 est une version ascendante du calcul de la récursion. Dans ce cas précis, il faut faire attention au fait que les appels récursifs accroissent l'indice du début des chaînes, donc pour que le calcul soit bien ascendant il faut commencer par les chaines d'indice de début maximum. Cela explique que la boucle de la ligne 10 soit décroissante

3.2.3 Arbres binaires de recherche optimaux

Étant donné un ensemble de mots S, nous souhaitons construire un dictionnaire sur S qui permettent de rechercher rapidement un mot de S, afin de récupérer une valeur qui lui est associée.

Plus spécifiquement, nous voulons coder S par un arbre binaire de recherche.

De plus, tous les mots ne seront pas recherchés avec la même fréquence. Pour chaque mot, nous connaissons la probabilité p (non-nulle) qu'une recherche concerne ce mot. Par exemple, nous pourrions avoir le dictionnaire suivant :

mot	probabilité p
chat	0.25
chien	0.3
hamster	0.15
poisson	0.2
tortue	0.1

Notre but est de trouver un arbre binaire de recherche, mais plutôt que le demander équilibré (comme nous le ferons dans le Chapitre **??**), nous souhaitons que la profondeur d'un nœud pris aléatoirement, selon les probabilités données dans la table, soit minimum. Formellement nous souhaitons minimiser sur les arbres binaires T, la fonction suivante :

$$\mathsf{cout}(T) = \sum_{\mathsf{mot} \in S} p(\mathsf{mot}) \cdot \mathsf{profondeur}_T(\mathsf{mot})$$

La première étape pour construire notre algorithme est de comprendre la structure des arbres minimisant notre objectif.

Lemme 3.2.5. *Si $T = (G, r, D)$ est un arbre binaire de recherche optimal, alors G et D sont des arbres binaires de recherche optimaux.*

Démonstration. Notons d'abord que le coût de T s'exprime en fonction du coût de G et D

$$\begin{aligned}
\mathsf{cout}(T) &= \sum_{\mathsf{mot} \in S} p(\mathsf{mot}) \cdot \mathsf{profondeur}_T(\mathsf{mot}) \\
&= p(r) \cdot 0 + \sum_{\mathsf{mot} \in G} p(\mathsf{mot}) \cdot \mathsf{profondeur}_T(\mathsf{mot}) + \sum_{\mathsf{mot} \in D} p(\mathsf{mot}) \cdot \mathsf{profondeur}_T(\mathsf{mot}) \\
&= \sum_{\mathsf{mot} \in G} p(\mathsf{mot}) \cdot (\mathsf{profondeur}_G(\mathsf{mot}) + 1) + \sum_{\mathsf{mot} \in D} p(\mathsf{mot}) \cdot (\mathsf{profondeur}_D(\mathsf{mot}) + 1) \\
&= \mathsf{cout}(G) + \mathsf{cout}(D) + \sum_{\mathsf{mot} \in S} p(\mathsf{mot}) - p(\mathsf{r})
\end{aligned}$$

S'il existe G' ayant les mêmes sommets que G avec $\mathsf{cout}(G') < \mathsf{cout}(G)$, alors pour $T' = (G', r, D)$, $\mathsf{cout}(T') = \mathsf{cout}(T) + \mathsf{cout}(G') - \mathsf{cout}(G) < \mathsf{cout}(G)$, contradiction. Donc G est optimal. Symétriquement, D est optimal. □

De plus, une fois fixé la racine r, G doit contenir tous les nœuds inférieurs à r, et D tous les nœuds supérieurs à r. Nous obtenons donc, en notant $\mathsf{OPT}(S')$ le coût minimum d'un arbre sur l'ensemble de mots $S' \subseteq S$ non vide :

$$\mathsf{OPT}(S') = \sum_{\mathsf{mot} \in S'} p(\mathsf{mot}) + \min_{r \in S'} \left(\mathsf{OPT}(\{\mathsf{mot} \in S' : \mathsf{mot} < r\}) + \mathsf{OPT}(\{\mathsf{mot} \in S' : \mathsf{mot} > r\}) - p(r)\right)$$

et en notant $T(S')$ l'arbre optimal pour S',

$$T(S') = (T(\{\mathsf{mot} \in S' : \mathsf{mot} < r\}), r, T(\{\mathsf{mot} \in S' : \mathsf{mot} > r\}))$$

où r est l'argument minimum dans la formule pour $\mathsf{OPT}(S')$. Enfin,

$$\mathsf{OPT}(\varnothing) = 0, \qquad T(\varnothing) = \bot$$

Nous obtenons donc une définition récursive de l'arbre optimal. Cette définition est valide : les appels se font sur les chaînes de la séquence des mots triés. Il y a donc $O(n^2)$ sous-problèmes

```
1   fonction arbre_optimal(tableau de chaîne mots, tableau de flottant p) : arbre de chaîne =
2     soit n := longueur(mots)
3     soit opt_table := tableau[1..n][1..n] de flottant ou bien ⊥
4     soit arbre_table := tableau[1..n][1..n] d'arbre binaire ou bien ⊥
5
6     fonction cout_découpe(entier i, entier j, entier k) : flottant =
7       retourner cout_optimal(i, j − 1) + cout_optimal(j + 1, k) − p(j)
8
9     fonction arbre_optimal(entier i, entier j) : arbre binaire =
10      si i > j alors retourner ⊥ sinon
11      si arbre_table[i][j] = ⊥ alors
12        soit r := argmin_{k∈[i,j]} cout_découpe(i, k, j)
13        opt_table[i][j] ← cout_découpe(i, r, j) + ∑_{k=i}^{j} p(k)
14        arbre_table[i][j] ← joint(arbre_table[i][r − 1], mots[r], arbre_table[r + 1][j])
15      retourner arbre_table[i][j]
16
17    fonction cout_optimal(entier i, entier j) : flottant =
18      si i > j alors retourner 0 sinon
19      soit calcule_l_arbre := arbre_optimal(i, j)
20      retourner opt_table[i][j]
21
22    retourner arbre_optimal(1, n)
```

FIGURE 3.9 – Calcul d'un arbre binaire de recherche optimal pour n mots triés. La probabilité de chaque mot est donné par le tableau p.

pour une instance avec n mots. Une version mémoïsée de l'algorithme est donné en Figure 3.9, qui utilise deux tableaux bidimensionnels pour stocker les solutions des sous-problèmes, l'un pour l'arbre et l'autre pour sa valeur.

Nous pouvons maintenant établir la complexité de cet algorithme. Nous comptons séparément la complexité générée par les lignes 10 à 12. Les lignes 10 à 12, exécutées une seule fois par case des tableaux opt_table et arbre_table, exigent de trouver un minimum parmi au plus n valeurs (ligne 10) (pour l'instant, nous ne comptons pas les appels de fonctions). La ligne 11 demande le calcul d'une somme sur au plus n valeurs, ainsi, chaque exécution des lignes 10 à 12, toujours sans compter les appels de fonctions, demande un temps de $O(n)$. Puisque ces 3 lignes sont exécutées n^2 fois, leur contribution totale est de $O(n^3)$. De plus cout_decoupe et cout_optimal sont appelés $O(n^3)$ fois en tout, chaque appel prenant un temps constant (hors coût des lignes 10 à 12). Donc la complexité pour les lignes autre que 10 à 12 est de $O(n^3)$ aussi. L'algorithme est donc de complexité $O(n^3)$

Exemple 3.2.1. En reprenant l'exemple du début de section, nous obtenons les tableaux de la Figure 3.10.

3.3 Complexité des algorithmes récursifs (*a.k.a.* diviser pour régner)

Un autre exemple bien connu d'algorithme récursif est le tri par fusion, qui prend une liste d'éléments (disons des entiers par exemple), et retourne la liste ordonnée de ces éléments. Dans ce cas, G_f est un graphe sur l'ensemble des listes d'entiers. Nous rappelons l'algorithme : étant donné une liste $L = \langle e_0, \ldots, e_n \rangle$, si $n \leq 0$, L est la liste triée, sinon nous trions les listes $L_0 = \langle e_0, e_2, \ldots \rangle$ et $L_1 = \langle e_1, e_3, \ldots \rangle$ par des appels récursifs, puis nous appliquons l'opération de fusion de listes triées sur L_1 et L_2.

À nouveau, chaque étape (sur une liste de longueur au moins 2) provoque deux appels récursifs. Toute la question est de savoir le nombre de sommets de G_f qui seront calculés. Un aperçu dans

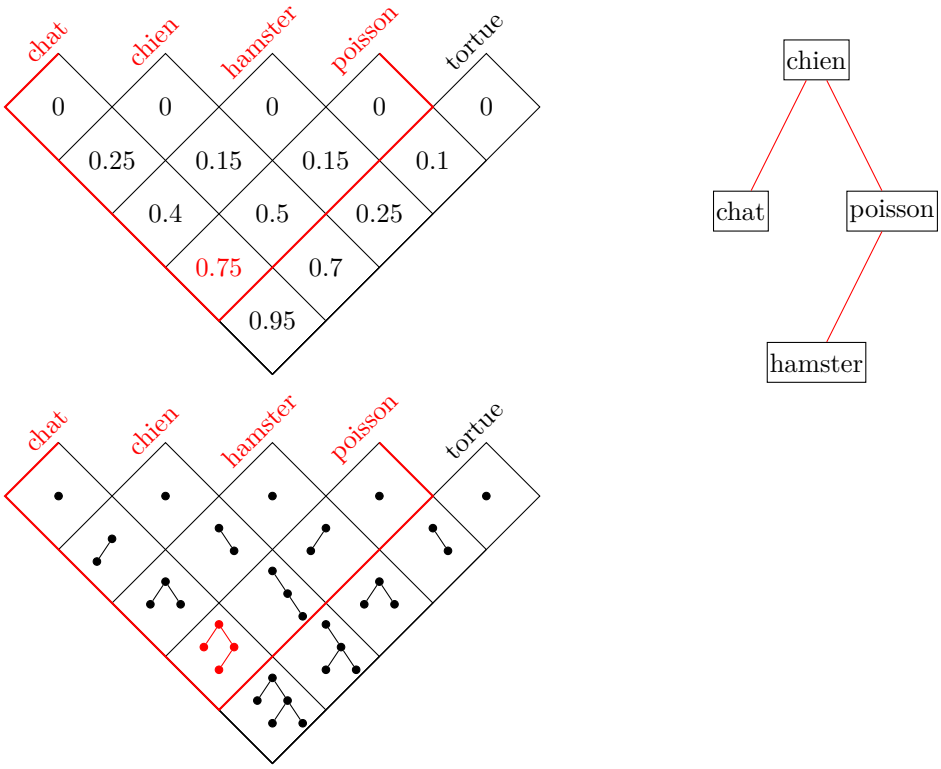


FIGURE 3.10 – Calcul de l'arbre optimum sur un exemple. Une des tables donne le coût optimal, l'autre donne l'arbre optimal (ici, seul la forme de l'arbre est donnée, l'arbre se déduit ensuite grâce à l'invariant des arbres binaires de recherche). Une case correspond à l'ensemble de mots de la pyramide pointée vers le bas, dont le sommet est cette case. À droite, l'arbre optimal pour les 4 premiers mots, figuré en rouge dans les tableaux.

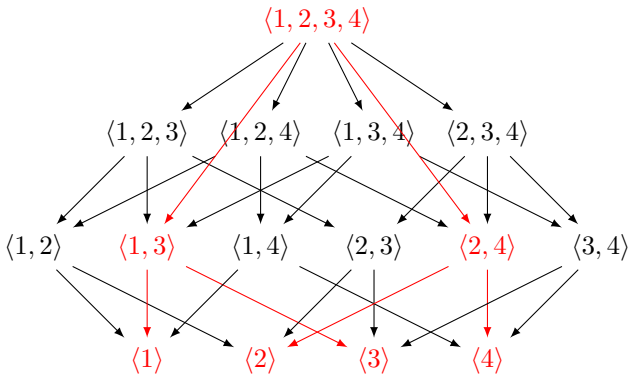


FIGURE 3.11 – Une toute petite partie de G_f pour le tri par fusion, et les sommets accessibles depuis $\langle 1, 2, 3, 4\rangle$ (tous les arcs ne sont pas dessinés).

le cas de la liste $\langle 1, 2, 3, 4\rangle$ est donné en Figure 3.11.

Ici, les sous-listes sur lesquels nous faisons des appels récursifs sont en général distinctes les unes des autres (c'est le cas si tous les éléments sont différents), donc la mémoïsation ne nous fait pas gagner de temps. Comme chaque évaluation fait jusqu'à deux appels récursifs, nous pouvons craindre à nouveau que le nombre d'appels explose exponentiellement. En fait ce n'est pas le cas, car la longueur du plus long chemin orienté partant d'une liste L dans G_f est approximativement $\log_2 \mathsf{longueur}(L)$: à chaque appel récursif, la longueur des listes est divisée par 2. Du coup le nombre total d'appels est $O(2^{\log_2 n}) = O(n)$, où n est le nombre d'éléments de L. Chaque opération prend un temps $O(n)$, ce qui donne en première approche une complexité de $O(n^2)$.

Ce calcul n'est pas très bon. La raison tient dans le fait que nous avons compté un temps $O(n)$ pour chaque sommet de G_f calculé, alors que la grande majorité des sommets calculés concerne des listes de taille bien plus petite que n. Nous pouvons donner une meilleure estimation en utilisant une technique générale, qui consiste à donner une formule récursive de la complexité de l'algorithme : puisque l'algorithme est récursif, calculons sa complexité avec une fonction définie récursivement.

Posons $g(n)$ la complexité maximale d'un tri par fusion d'une liste de longueur n. Par complexité maximale, nous entendons le maximum du nombre d'opérations effectuées, pris sur toutes les listes de longueur n (c'est donc un pire des cas). Nous savons que pour une liste de longueur n, l'algorithme fait deux appels récursifs sur des listes de longueurs $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$, et une opération de fusion de complexité $O(n)$ (plus de l'administration, coûtant $O(1)$). Nous pouvons donc écrire :

$$g(n) = g\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + g\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n) + O(1)$$

En pratique, il n'est pas nécessaire d'être aussi précis sur les arrondis, nous nous contentons donc de :

$$g(n) = 2 \cdot g\left(\frac{n}{2}\right) + \Theta(n)$$

Pour résoudre cette équation, nous utilisons le théorème suivant (que nous admettons) :

Théorème 3.3.1. *Soit $a \geq 1$ et $b > 1$, f une fonction et T définie par la relation de récurrence :*

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

- *Si $f(n) = O(n^{\log_b a - \epsilon})$ pour un $\epsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.*
- *Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \cdot \log n)$.*
- *Si $f(n) = \Omega(n^{\log_b a + \epsilon})$ pour un $\epsilon > 0$, et asymptotiquement $a \cdot f(n/b) \leq c \cdot f(n)$ pour un $c < 1$, alors $T(n) = \Theta(f(n))$*

```
1   fonction partition(t pivot, liste de t liste) : (liste de t, liste de t) =
2     si est_vide(liste) alors retourner (⟨⟩, ⟨⟩)
3     soit (liste_inf, liste_sup) := partition(pivot, queue(liste))
4     si tete(liste) ≤ pivot alors
5       retourner (insertion(tete(liste), liste_inf), liste_sup)
6     sinon retourner (liste_inf, insertion(tete(liste), liste_sup))
7
8   fonction selection(entier k, liste de t liste) : t =
9     si longueur(liste) ≤ 1 alors
10      retourner tete(liste)
11    soit pivot ∈ liste
12    soit (liste_inf, liste_sup) := partition(pivot, liste)
13    si longueur(liste_inf) > k alors
14      retourner selection(k, liste_inf)
15    sinon retourner selection(k − longueur(liste_inf), liste_sup)
```

FIGURE 3.12 – Sélection du k^{e} élément.

Pour le tri par fusion, $a = 2$ et $b = 2$, donc $\log_b a = 1$, nous sommes dans le deuxième cas, et la complexité du tri par fusion est donc $O(n \log n)$.

3.3.1 Sélection du k^{e} élément

Soit liste $= \langle e_1, \ldots, e_n \rangle$ une séquence de n éléments distincts d'un ensemble ordonné. Le problème de la *sélection du k^e élément* (pour $k \in [1, n]$) consiste à trouver l'élément e_i tel que $|\{e_j : j \in [1, n], e_j \leq e_i\}| = k$, autrement dit le k^{e} plus petit élément de la séquence. Si $k = \lceil n/2 \rceil$, nous parlons de problème du *médian*, c'est-à-dire l'élément qui a autant d'éléments plus grands que de plus petits (à un près). Nous appelons *rang* d'un élément e d'une liste $\langle e_1, \ldots, e_n \rangle$ la quantité $|\{e_j : j \in [1, n], e_j \leq e\}|$. Nous cherchons donc l'élément de rang k.

Une solution simple consiste à trier la séquence en une séquence croissante, puis à prendre le k^{e} élément de la séquence triées. Clairement cette solution a une complexité dominée par le tri, en $\Theta(n \log n)$.

Notre objectif est de trouver une solution en complexité linéaire $\Theta(n)$. Pour cela, nous nous inspirons de l'algorithme de tri rapide pour donner un algorithme randomisé. Nous prenons un pivot, puis partitionnons la liste en deux listes, celles des éléments inférieurs ou égaux au pivot, notée list_inf, et celle des éléments supérieurs notée liste_sup.

Si |liste_inf| $> k$, le k^{e} élément de liste est aussi le k^{e} élément de liste_inf. Sinon, liste_sup contient le k^{e} élément de liste, mais possède k éléments inférieurs au k^{e} élément de moins que liste, l'élément recherché est donc le $k - |\text{liste\_inf}|^{\text{e}}$ élément de liste_sup. La Figure 3.12 récapitule l'algorithme que nous venons de décrire.

La seule partie non précisée dans ce code concerne le choix du pivot. Considérons dans un premier temps que le pivot est choisi aléatoirement uniformément parmi tous les éléments de la liste, et faisons une analyse de complexité en espérance pour cet algorithme. Notons d'abord que partition a pour complexité $\Theta(n)$ sur une liste de longueur n (chaque appel récursif diminue de 1 la longueur de la liste en argument). La sélection d'un pivot prend aussi un temps linéaire : il suffit de tirer un entier i et de prendre le i^{e} élément de la liste. Ainsi, la complexité hors appel récursif est de $\Theta(n) \leq b \cdot n$ (pour une constante b bien choisie) pour une liste de taille n. Pour les appels récursifs, nous distinguons deux cas : le pivot est de rang r supérieur ou égal à k (appel récursif ligne 14, sur une liste de longueur r), ou bien le pivot est de rang r inférieur à k (appel récursif ligne 15, sur une liste de longueur $n - r$). Nous obtenons alors une formule pour $T(n)$, la

complexité en espérance pour une liste de longueur n :

$$\begin{aligned} T(n) &\leq b \cdot n + \sum_{r=k}^{n} \frac{1}{n} T(r) + \sum_{r=1}^{k-1} \frac{1}{n} T(n-r) \\ &\leq b \cdot n + \sum_{r=1}^{n} \frac{1}{n} T(\max\{r, n-r\}) \\ &\leq b \cdot n + \frac{2}{n} \sum_{r=\lfloor n/2 \rfloor}^{n} T(r) \end{aligned}$$

Nous conjecturons que $T(n) \leq c \cdot n$, pour une constante c supérieure à $5 \cdot b$. Pour cela nous le démontrons par récurrence, en supposant que c'est vrai pour tout $n' < n$ et en le démontrant pour n. De fait, pour n nous avons alors :

$$\begin{aligned} T(n) &\leq b \cdot n + \frac{2}{n} \sum_{r=\lfloor n/2 \rfloor}^{n} c \cdot r \\ &\leq b \cdot n + \frac{2 \cdot c}{n} \cdot \frac{(n - \lfloor n/2 \rfloor + 1)(n + \lfloor n/2 \rfloor)}{2} \\ &\leq b \cdot n + \frac{c \cdot \left(n^2 + n + n/2 + 1 - (n/2+1)^2\right)}{n} \\ &\leq b \cdot n + c \cdot \left(n + 1 + \frac{1}{2} - \frac{n}{4} - \frac{1}{n} + 1\right) \\ &\leq b \cdot n + c \cdot \left(\frac{3n}{4} + \frac{5}{2}\right) \leq c \cdot n \end{aligned}$$

pour n assez grand (les cas où n est petit sont immédiats en choisissant c suffisamment grand).

Donc cette version randomisée a une complexité $\Theta(n)$ en espérance. Nous allons maintenant améliorer le choix du pivot, pour produire un algorithme déterministe (sans utilisation d'aléa) de complexité $\Theta(n)$ dans le pire des cas. Il s'agit uniquement d'un exercice : en pratique, comme pour le tri rapide, l'algorithme randomisé de sélection offre des performances supérieures à la version déterministe.

Nous choisissons le pivot par la méthode suivante. Les éléments de la liste sont groupés par paquets de 5 (un des paquets peut contenir moins de 5 éléments, si n n'est pas multiple de 5). Nous établissons ensuite la liste liste_medians des médians de chaque paquet, et nous cherchons récursivement le médian de liste_medians, qui sera notre pivot. L'algorithme est décrit en Figure 3.13, et une illustration est donné en Figure 3.14.

Nous analysons maintenant cet algorithme.

Lemme 3.3.1. *Le rang de l'élément p retourné par* choix_pivot(liste) *est compris entre* $3n/10 - 2$ *et* $7n/10 + 2$*, avec* $n =$ longueur(liste)*.*

Démonstration. p est le médian de la liste liste_medians de longueur $m = \lceil \frac{n}{5} \rceil$ des médians, donc au moins $\lfloor \frac{m+1}{2} \rfloor$ médians sont inférieurs ou égaux à p. Chacun de ces médians possèdent deux éléments qui lui sont plus petits dans le paquet de 5 éléments dont il provient (moins 2 pour tenir compte du paquet faisant moins de 5 éléments), chacun de ces éléments est donc inférieur à p par transitivité. Nous avons donc que le nombre d'éléments inférieurs ou égaux à p est au moins :

$$3 \times \left\lfloor \frac{\lceil \frac{n}{5} \rceil + 1}{2} \right\rfloor - 2 \geq \frac{3 \lceil \frac{n}{5} \rceil}{2} - 2 \geq \frac{3n}{10} - 2$$

L'analyse est la même pour compter des éléments supérieurs à p, au moins $\frac{3n}{10} - 2$ éléments sont supérieurs ou égaux à p. Les bornes sur le rang de p s'en déduisent. □

```
1     // retourne le kᵉ élément d'une liste.
2   fonction element(entier k, liste de t liste) : t =
3     si k = 1 alors retourner tete(liste)
4     sinon retourner element(k − 1, queue(liste))
5
6     // retourne le préfixe de longueur k d'une liste, et le suffixe restant.
7   fonction couper(entier k, liste de t liste) : (liste de t, liste de t) =
8     si k = 0 alors retourner (⟨⟩, liste)
9     sinon soit (prefixe, suffixe) := couper(k − 1, queue(liste))
10      retourner (insertion(tete(liste), prefixe), suffixe)
11
12  fonction median_naif(liste de t liste) : t =
13    retourner element((longueur(liste) + 1)/2, trier(liste))
14
15  fonction extraire_medians(liste de t liste) : liste de t =
16    si longueur(liste) < 5 alors
17      retourner ⟨median_naif(liste)⟩
18    sinon
19      soit (prefixe, suffixe) := couper(5, liste)
20      retourner insertion(median_naif(prefixe), extraire_medians(suffixe))
21
22  fonction choix_pivot(liste de t liste) : t =
23    soit liste_median := extraire_median(liste)
24    retourner selection((1 + longueur(liste_median))/2, liste_median)
```

FIGURE 3.13 – Choisir un bon pivot pour le problème de la sélection. Cela permet aussi de déterminiser l'algorithme de tri rapide.

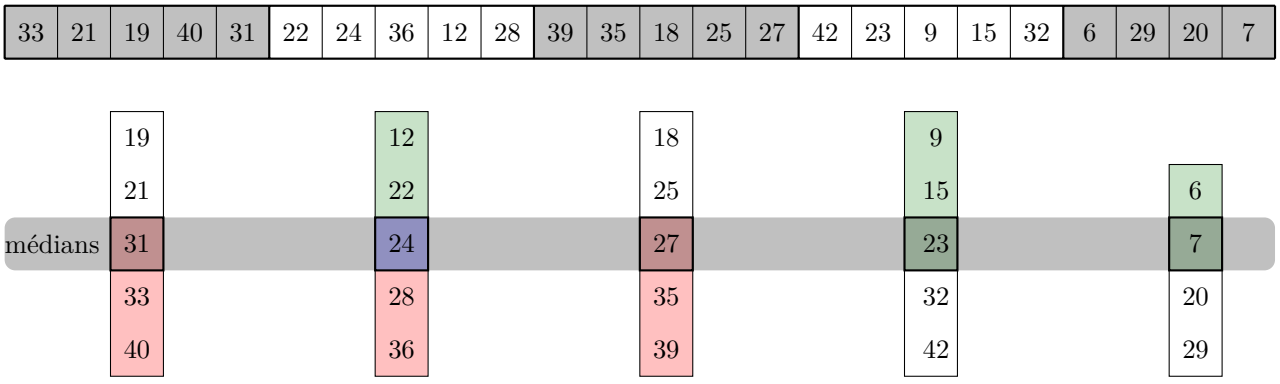


FIGURE 3.14 – Un exemple de sélection du pivot. La liste est coupée en morceaux de 5 (représentés ici verticalement) et chaque morceau de 5 est trié. Le médian des médians de chaque morceau est 24. La moitié des morceaux de 5 possède la moitié de leurs éléments supérieurs au pivot (en rouge), l'autre moitié possède une moitié de leurs éléments inférieurs au pivot (en vert).

Ainsi le pivot se trouve approximativement au milieu de la liste en terme de rang. Nous pouvons maintenant calculer la complexité de l'algorithme de sélection ainsi obtenu. Notons dans un premier temps la complexité de choix_pivot, sans compter l'appel récursif à selection : les fonctions element et couper ont une complexité $\Theta(k)$, et nous les appelons avec $k = 5$, donc pour notre usage, leur complexité est $O(1)$. median_naif a une complexité dominée par le tri, mais à nouveau nous l'utilisons seulement sur des listes de taille au plus 5, donc chaque appel prend un temps constant $O(1)$ aussi. extraire_median est une fonction récursive, qui hormis l'appel récursif prend un temps constant. De plus la longueur de son argument décroit strictement à chaque appel récursif, donc le nombre total d'appels récursifs est linéaire en la longueur de la liste, sa complexité s'établit donc à $\Theta(n)$ pour une liste de longueur n.

Nous retournons maintenant à l'analyse de selection. Hormis le choix du pivot et l'appel récursif, la complexité est toujours de $\Theta(n)$. Le choix du pivot prend $\Theta(n)$ aussi, plus un appel récursif sur une liste de longueur $\frac{n}{5}$. L'appel récursif final s'effectue maintenant sur une liste d'au plus $\frac{7n}{10} + 2$ éléments, ce qui nous donne pour la complexité $P(n)$ dans le pire des cas sur une liste de longueur n :

$$P(n) \leq P\left(\frac{n}{5}\right) + P\left(\frac{7n}{10}\right) + \Theta(n)$$

Il existe une constante b telle que :

$$P(n) \leq P\left(\frac{n}{5}\right) + P\left(\frac{7n}{10}\right) + b \cdot n$$

Nous ne pouvons pas utiliser le théorème général pour une récurrence de cette forme. Nous utilisons donc la méthode de substitution (que nous avons déjà utilisée pour l'analyse en espérance) : nous conjecturons que $P(n) \leq c \cdot n$ pour une certaine constante n (que nous déterminerons plus tard). La preuve se fait par récurrence. Pour n petit, il suffit de choisir c assez grand. Dans le cas général, supposons que notre conjecture soit vraie pour tout entier inférieur à n. Nous substituons :

$$\begin{aligned} P(n) &\leq c \cdot \frac{n}{5} + c \cdot \frac{7n}{10} + b \cdot n \\ &\leq \left(\frac{9}{10} \cdot c + b\right) \cdot n \\ &\leq c \cdot n \end{aligned}$$

Pour que la dernière inégalité soit correcte, nous choisissons de prendre $c \geq 10 \cdot b$. Par récurrence, la complexité est bien $P(n) = O(n)$.

3.3.2 Recherche des deux points les plus proches

Considérons un ensemble de points du plan Euclidien (x_i, y_i) pour $i \in [\![1, n]\!]$. Comment trouver le plus efficacement possible la paire de points les plus proches ? Formellement, trouver i et j minimisant l'expression :

$$d_{i,j} := \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

La solution naïve consiste à tester toutes les paires (i, j) et à garder la meilleure. Le nombre de tests est donc $n(n-1)/2$, et la complexité de cet algorithme est $O(n^2)$.

Nous utilisons maintenant un algorithme de type *diviser-pour-régner* pour obtenir une complexité de $O(n \log n)$.

La première étape est de couper le problème en deux. Dans ce type d'algorithmes, il est généralement plus efficace de couper en deux problèmes de tailles égales. Ici, nous aimerions avoir autant de points (à un près) dans chaque sous-problème. Simplement, nous allons prendre les $\lfloor n/2 \rfloor$ points de plus petites abscisses pour définir le premier sous-problème (gauche), et les $n - \lfloor n/2 \rfloor \approx n/2$ points d'abscisses les plus grandes pour le second sous-problème (droit). Pour

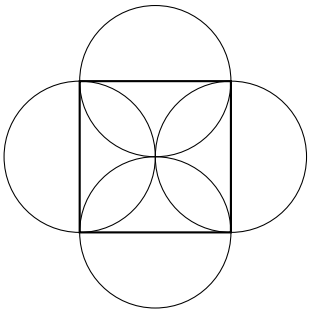


FIGURE 3.15 – Couverture d'un carré de côté 1 par 4 disques de diamètre 1

cela, il suffit de trier les points par abscisses croissantes. Nous pouvons aussi noter x^0 l'abscisse maximum d'un point de gauche : la droite verticale Δ^0 d'abscisse x^0 sépare donc les points de gauche des points de droite.

Par récurrence, nous déterminons les deux points à gauche les plus proches et les deux points à droite les plus proches. La meilleure des deux paires nous donne un candidat (i, j) comme paire de points la plus proche, avec une distance $d := d_{i,j}$.

Cette paire (i, j) est alors la paire de points la plus proche parmi tous les points, sauf s'il existe un point à gauche i' et un point à droite j' tels que $d_{i',j'} < d$. Il nous faut donc vérifier si une telle paire existe. Malheureusement, le nombre de paires possibles est à nouveau $O(n^2)$, donc les tester naïvement n'est pas efficace.

Nous devons donc réduire le nombre de tests entre points à gauche et points à droite. Pour cela, nous utilisons le fait que nous connaissons d et que seules les paires à distance moins que d nous intéressent. Cela exclut immédiatement certains points :

— les points de gauche d'abscisses strictement inférieures à $x^0 - d$ sont à distances supérieures à d du côté droit du plan, donc de point point de droite.

— les points de droite d'abscisses strictement supérieures à $x^0 + d$ sont disqualifiés pour la même raison

Malheureusement, tous les points peuvent être dans la bande de largeur $2d$ autour de la droite verticale Δ^0. Cette remarque seule ne peut donc suffire à améliorer la complexité. Nous utilisons donc l'observation supplémentaire suivante : si $d_{i',j'} < d$ alors $|y_{i'} - y_{j'}| \leq d$.

Lemme 3.3.2. *Pour tout ensemble P de points $(x_i, y_i)_i$, avec $d := \min_{i \neq j} d_{i,j}$, tout carré de côté d contient au plus 4 points de P.*

Démonstration. Toute surface de diamètre d contient au plus un point de P. Un carré de côté d est contenu dans l'union de 4 disques de diamètre d (cf Figure 3.15), d'où le résultat. □

Lemme 3.3.3. *Soit P un ensemble de points $(x_i, y_i)_i$, $x^0 \in \mathbb{R}$ et*

$$d := \min\left(\min\{d_{i,j} \mid x_i \leq x^0 \wedge x_j \leq x^0\}, \min\{d_{i,j} \mid x_i \geq x^0 \wedge x_j \geq x^0\}\right)$$

Soit i, j minimisant $d_{i,j}$, avec $y_i \leq y_j$, alors

$$\left|\{(x_k, y_k) \in P \mid x_k \in [x^0 - d, x^0 + d], y_k \in [y_i, y_j]\}\right| \leq 8$$

Démonstration. Il suffit de considérer les carrés de côtés d et de coins inférieurs gauches $(x^0 - d, y_i)$ et (x^0, y_i). Comme $y_j - y_i \leq d_{i,j} \leq d$, tous les points de l'ensemble sont contenus dans un de ces deux carrés. De plus chacun des deux carrés ne contient que des points à distance au moins d les uns des autres. Par le Lemme 3.3.2, l'ensemble contient bien au plus 8 points. □

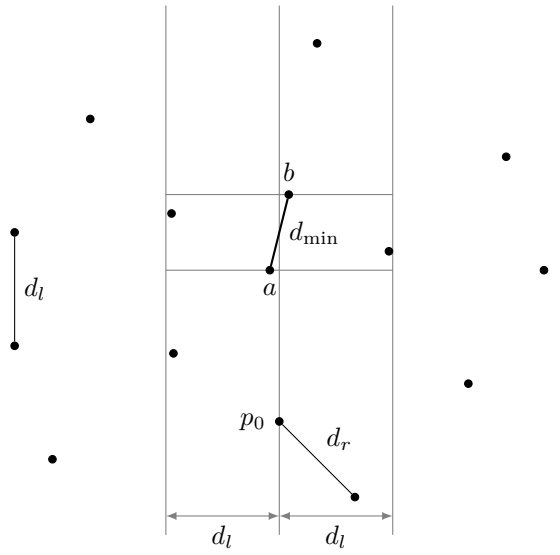


FIGURE 3.16 – Exemple de l'algorithme de recherche de la paire de points la plus proche. p_0 est le sommet médian en terme d'abscisse. d_l et d_r sont les plus courtes distances respectivement sur les points à gauche et à droite de p_0. $d_{\min}$ est la plus petite distance dans l'absolu. La bande verticale de largeur $2\min\{d_l, d_r\}$ centrée en p_0 limite le nombre de paires à tester : a et b sont séparés verticalement par seulement 2 autres points dans cette bande (au maximum 6 dans le pire des cas).

Ce Lemme 3.3.3 nous dit que si la paire de points la plus proche est séparée par la droite verticale Δ^0, alors ces deux points ne possèdent qu'au plus six éléments les séparant dans la liste triée par ordonnées croissantes des points dans la bande autour de Δ^0. Le nombre de paires à tester passe donc à au plus $7n = O(n)$.

Cette ébauche d'algorithme nous donne une complexité $c : \mathbb{N} \to \mathbb{R}^+$ qui satisfait la formule :

$$c(n) = 2.c(n/2) + \alpha.n.\log n + \beta.n$$

Cette formule donne seulement une complexité de $c(n) = O(nlog^2 n)$. Pour pouvoir atteindre la complexité que nous nous sommes fixés, il faut supprimer le terme en $n \log n$ de la formule. Celui-ci provient du tri des points par abscisses croissantes et par ordonnées croissantes. Pour cela, il suffit de remarquer que nous n'avons besoin de trier les points qu'une seule fois au début de l'algorithme, selon les deux coordonnées. Nous obtenons alors une complexité de $O(n \log n) + c(n)$ avec :

$$c(n) = 2.c(n/2) + \beta.n$$

ce qui se résout en $O(n \log n)$.

Chapitre 4

Approximation

On s'intéresse dans ce chapitre à des problèmes d'optimisation. Une instance I d'un *problème d'optimisation* est décrit par une fonction objectif f définie sur un ensemble de solutions admissibles S à un problème : l'objectif est de calculer une solution admissible $s \in S$ qui minimise ou maximise (selon le cas) la fonction objectif f. On parle de *solution optimale* et on la note souvent OPT, ou $\mathrm{OPT}(I)$.

Il est souvent très difficile de trouver une solution optimale. Dans ce cas-là, on se contente souvent de trouver une solution qui approche l'optimum. Étant donné un problème de minimisation (respectivement, de maximisation) et un nombre réel positif $\delta \geq 1$ (respectivement, $\delta \leq 1$), un algorithme $\mathcal{A}$ est un *algorithme d'approximation avec un facteur* δ si pour chaque instance I du problème, l'algorithme $\mathcal{A}$ produit une solution admissible s telle que $f(s) \leq \delta \cdot \mathrm{OPT}(I)$ (respectivement, $f(s) \geq \delta \cdot \mathrm{OPT}(I)$).

4.1 Ordonnancement des tâches (*scheduling*, ou *load balancing*)

On se donne m machines identiques $M_1, M_2, \ldots, M_m$, ainsi que n tâches de durées $p_1, p_2, \ldots, p_n$ qui doivent être toutes exécutées dans n'importe quel ordre. Chaque machine ne peut exécuter qu'une tâche à la fois, sans pouvoir arrêter une tâche au milieu de son exécution. Chaque tâche doit donc être affectée à une unique machine. Une affectation des tâches sur les machines consiste donc à attribuer à chaque machine M_i, pour $i \in \{1, \ldots, m\}$, une liste des tâches L_i qu'elle doit exécuter dans l'ordre. Un ordonnancement est donc un n-uplet de listes $(L_1, \ldots, L_m)$ contenant des éléments dans $\{1, \ldots, n\}$ tel que chaque tâche $j \in \{1, \ldots, n\}$ apparaisse dans une et une seule de ces listes. Le temps d'exécution de la machine M_i est alors la somme des durées des tâches qu'elle doit exécuter : $t_i = \sum_{j \in L_i} p_j$. Le temps total d'exécution de l'ordonnancement est donc $\max_{1 \leq i \leq m} t_i$: c'est la fonction objectif du problème, les solutions admissibles de l'instance du problème étant tous les ordonnancements possibles. Le problème d'ordonnancement consiste à trouver une affectation des n tâches aux m machines telle que le temps total d'exécution (*makespan*) soit minimum.

Considérons un exemple, avec 3 machines et 6 tâches de durées respectives $p_1 = 2$, $p_2 = 3$, $p_3 = 4$, $p_4 = 6$, $p_5 = 2$, $p_6 = 2$. Un ordonnancement possible consiste à exécuter les tâches 1 et 4 sur la machine M_1, les tâches 2 et 3 sur la machine M_2 et les tâches 5 et 6 sur la machine M_3 : on a donc $L_1 = [1, 4]$, $L_2 = [2, 3]$, $L_3 = [5, 6]$. Le temps total d'exécution de cet ordonnancement est donc $\max(t_1, t_2, t_3)$ avec $t_1 = 8$, $t_2 = 7$ et $t_3 = 4$: c'est donc 8. Il existe un meilleur ordonnancement de temps total d'exécution 7 qui répartit mieux la charge entre les machines : $L_1 = [4]$, $L_2 = [2, 3]$, $L_3 = [1, 5, 6]$.

Un premier algorithme simple consiste à considérer les tâches dans l'ordre croissant, et à exécuter la tâche courante sur la machine la moins utilisée jusqu'ici, c'est-à-dire sur la machine avec un temps d'exécution courant minimum. Il s'agit d'un algorithme glouton, qu'on appelle algorithme LSA (pour *List Scheduling Algorithm*) : il prend une décision avec les informations

dont il dispose à un moment donné de son exécution et ne revient jamais sur cette décision dans le futur. Sur l'instance précédente, cet algorithme affecte

— la tâche 1 à la machine M_1,
— la tâche 2 à la machine M_2,
— la tâche 3 à la machine M_3,
— la tâche 4 à la première machine qui termine, donc la machine M_1,
— la tâche 5 à la machine qui termine la plus vite ensuite, donc la machine M_2,
— et la tâche 6 à la machine qui termine la plus vite finalement, à savoir la machine M_3.

On obtient ainsi l'ordonnancement $L_1 = [1, 4]$, $L_2 = [2, 5]$, $L_3 = [3, 6]$ de temps total d'exécution 8 non optimal. L'algorithme ne produit donc pas un ordonnancement optimal, mais on montre qu'il fournit cependant une approximation.

Théorème 4.1.1. *L'algorithme LSA est un algorithme d'approximation avec un facteur 2 pour le problème d'ordonnancement de tâches.*

Démonstration. Soit OPT le temps optimal pour une instance donnée et soit T le temps total d'exécution de l'ordonnancement produit par l'algorithme LSA. On peut donner deux bornes inférieures pour OPT :

— le temps moyen $(\sum_{i=1}^{n} p_i)/m$ de fonctionnement d'une machine ;
— la durée $\max_{1 \leq i \leq n} p_i$ de la tâche la plus longue.

Soit M_i la machine dont le temps d'exécution est égal à T. Soit j la dernière tâche exécutée sur M_i. Observons le comportement de l'algorithme à l'instant début$_j$ où il va affecter la tâche j. Comme l'algorithme l'affecte à la machine la moins occupée (c'est-à-dire celle qui termine le plus vite), toutes les machines sont occupées à cet instant debut$_j$: on sait donc que $m \cdot$début$_j \leq \sum_{i=1}^{j-1} p_i$. Alors,

$$\text{début}_j \leq \frac{1}{m} \sum_{i=1}^{j-1} p_i \leq \frac{1}{m} \sum_{i=1}^{n} p_i \leq \text{OPT}\,.$$

Puisque $p_j \leq \text{OPT}$, on obtient

$$T = \text{debut}_j + p_j \leq \text{OPT} + \text{OPT} = 2 \cdot \text{OPT}\,.$$

□

Le pire exemple pour l'algorithme LSA est constitué par m^2 tâches de durée 1 suivies par une tâche de durée m. L'ordonnancement obtenu par l'algorithme LSA est de temps total d'exécution $2m$, alors que OPT$=m+1$.

Un autre algorithme consiste à prétraiter la liste des tâches en les triant par ordre décroissant de leur temps d'exécution. On applique ensuite l'algorithme glouton LSA. On l'appelle l'algorithme LPT (pour *Largest Processing Time*) : c'est un meilleur algorithme d'approximation comme le montre le résultat suivant.

Théorème 4.1.2. *L'algorithme LPT est un algorithme d'approximation avec un facteur 4/3 pour le problème d'ordonnancement des tâches.*

Démonstration. Soit $I = (p_1 \geq p_2 \geq \cdots \geq p_n)$ la liste trié des tâches. Comme dans le Théorème 4.1.1, soit M_i la machine dont le temps de fonctionnement est égal au temps total T de l'ordonnancement produit par l'algorithme LPT et soit j la dernière tâche exécutée sur M_i.

— **Cas 1 :** $p_j \leq \text{OPT}/3$. En raisonnant comme dans la preuve du Théorème 4.1.1, on déduit que

$$\text{T} \leq \frac{1}{m} \sum_{i=1}^{j-1} p_i + p_j \leq \text{OPT} + \text{OPT}/3 = \frac{4}{3}\text{OPT}\,.$$

— **Cas 2 :** $p_j > \text{OPT}/3$. Considérons l'instance $I' = (p_1 \geq p_2 \geq \cdots \geq p_j)$. Soit OPT$'$ le temps optimal et soit T' le temps d'exécution des tâches de I' par l'algorithme LPT. Alors $T' = T$ (puisque la tâche j sera affectée à la même machine M_i par l'algorithme LPT) et

OPT′≤OPT. Il suffit donc de montrer que T'=OPT′ pour conclure. Or, chaque tâche de I' a une durée supérieure strictement à OPT/3. Chaque machine ne peut donc exécuter que deux tâches au maximum : dans ce cas-là, il n'est pas très difficile de montrer que l'algorithme LPT produit nécessairement un ordonnancement optimal ce qui permet de conclure. □

4.2 Sac à dos (*Knapsack*)

Le problème du sac à dos consiste en un ensemble de n objets, chacun ayant un poids p_i et une valeur v_i ($1 \leq i \leq n$), qu'on souhaite emporter dans un sac de capacité totale P (tous les nombres sont des entiers positifs). On cherche à trouver un sous-ensemble d'objets $S \subseteq \{1, 2, \ldots, n\}$ de poids total $\sum_{i \in S} p_i \leq P$ et de valeur $\sum_{i \in S} v_i$ maximale. Là encore, il s'agit d'un problème difficile à résoudre de manière exacte. Il est cependant facile d'en trouver une approximation de facteur 1/2 (c'est-à-dire qu'on trouvera une solution de valeur au moins la moitié de la valeur optimale) à l'aide de l'algorithme glouton suivant :

Algorithme SacàDosGlouton

1. `Trier les objets par l'ordre décroissant des rapports` v_i/p_i
2. `Trouver` $V_{\max} := \max\{v_i \mid i = 1, \ldots, n\}$
3. $Sac := \emptyset,\ p := P$
4. `pour` $i := 1$ `à` n `faire`
   `si` $p_i \leq p$ `alors` $Sac := Sac \cup \{i\}$; $p := p - p_i$
5. `Retourner` $ValeurGlouton = \max\{\sum_{i \in Sac} v_i, V_{\max}\}$

Théorème 4.2.1. *L'algorithme SacàDosGlouton est un algorithme d'approximation avec un facteur* 1/2*.*

Démonstration. Soit j l'indice du premier objet qui n'est pas pris dans Sac. La valeur renvoyée par l'algorithme jusqu'à ce point est $\text{valeur}_j = \sum_{i=1}^{j-1} v_i \leq \sum_{i \in Sac} v_i$ et son poids est $\text{poids}_j = \sum_{i=1}^{j-1} p_i \leq P$. On va montrer ensuite que OPT< $\text{valeur}_j + v_j$. Comme les objets sont ordonnés dans l'ordre décroissant en fonction du rapport valeur/poids, si on échange un sous-ensemble d'objets $\{1, \ldots, j-1\}$ avec un sous-ensemble admissible quelconque d'objets $\{j, j+1, \ldots, n\}$, la valeur globale ne va pas augmenter. Donc, la valeur optimale OPT est bornée par valeur_j plus la plus grande valeur obtenue en remplissant l'espace libre restant (c'est-à-dire $p - p_j$) par des objets avec un rapport valeur/poids inférieur à v_j/p_j. Comme $\text{poids}_j + p_j > p$, on obtient

$$\text{OPT} \leq \text{valeur}_j + (p - p_j) \cdot (v_j/p_j) < \text{valeur}_j + v_j\,.$$

Si $v_j \leq \text{valeur}_j$, alors

$$\text{OPT} \leq 2 \cdot \text{valeur}_j \leq 2 \cdot \sum_{i \in Sac} v_i\,.$$

Si $v_j > \text{valeur}_j$, alors $V_{\max} > \text{valeur}_j$, donc

$$\text{OPT} \leq \text{valeur}_j + v_j \leq \text{valeur}_j + V_{\max} < 2 \cdot V_{\max} \leq 2 \cdot ValeurGlouton\,.$$

Ainsi, dans tous les cas, on a bien $ValeurGlouton \geq \text{OPT}/2$. □

4.3 Remplissage des boîtes (*bin packing*)

On se donne un ensemble $E = \{e_1, \ldots, e_n\}$ de n objets, chacun ayant une taille s_i telle que $0 < s_i < 1$. On souhaite ranger tous les objets dans un minimum de boîtes de taille unitaire. Chaque boîte peut contenir tout sous-ensemble d'objets dont la taille globale n'excède pas 1. C'est donc encore un problème d'optimisation où la fonction objectif est le nombre de boîtes utilisées dans une solution pour ranger les n objets. Considérons un exemple avec 5 objets de taille $s_1 = 0.6$,

$s_2 = 0.7$, $s_3 = 0.2$, $s_4 = 0.4$ et $s_5 = 0.1$. Il est possible de ranger tous les objets dans deux boîtes en mettant les objets e_1 et e_4 dans une boîte et les objets e_2, e_3 et e_5 dans une autre boîte. Trouver le remplissage dans un nombre minimum de boîtes est un problème difficile dont on essaie donc de trouver une bonne approximation.

Un premier algorithme est l'algorithme Nextfit, un algorithme glouton : il consiste à prendre chaque objet l'un après l'autre et le placer dans la dernière boîte si on peut, ou à le placer dans une nouvelle boîte sinon. Dans l'exemple précédent, cela revient à ne placer que l'objet e_1 dans la première boîte, puis les objets e_2 et e_3 dans une seconde boîte et finalement les objets e_4 et e_5 dans une troisième boîte. L'algorithme ne produit donc pas un remplissage optimal des boîtes. Cependant, c'est bien un algorithme d'approximation :

Théorème 4.3.1. *NextFit est un algorithme d'approximation avec un facteur 2.*

Démonstration. Soit $S = \sum_{i=1}^{n} s_i$. On peut facilement voir que le nombre optimal de boîtes nécessaires est au moins $\lceil S \rceil$. D'autre part, NextFit utilise au maximum $2\lceil S \rceil$ boîtes : chaque paire de boîtes consécutives est remplie au moins à la moitié (donc d'au moins une unité en tout). □

Le pire exemple est une instance de $4n$ objets de tailles $(1/2, 1/2n, 1/2, 1/2n, \ldots, 1/2, 1/2n)$. On a alors un remplissage en OPT$=n+1$ boîtes, alors que NextFit utilise $2n$ boîtes.

Un autre algorithme consiste en l'algorithme FirstFit (de Fischer-Price) : il s'agit de prendre chaque objet l'un après l'autre et de le placer dans la première boîte qui peut l'accueillir. Dans l'exemple du début du chapitre, on ne trouve toujours pas un remplissage optimale, mais on peut montrer que le facteur d'approximation ainsi obtenu est meilleur : l'algorithme FirstFit est un algorithme d'approximation avec un facteur 1.7. On peut encore l'améliorer en triant préalablement les objets par ordre décroissant de tailles : il s'agit de l'algorithme FirstFitDecreasing. On suppose donc que les objets sont triés de sorte que $s_1 \geq s_2 \geq \ldots \geq s_n$ et on exécute ensuite l'algorithme FirstFit précédent. Sur l'exemple initial, cela revient à considérer les objets dans l'ordre $(e_2, e_1, e_4, e_3, e_5)$: on obtient alors le remplissage optimal. Ce n'est cependant pas toujours le cas, même si on obtient le meilleur résultat d'approximation suivant :

Théorème 4.3.2. *FirstFitDecreasing est un algorithme d'approximation avec un facteur 1.5 (plus une boîte).*

Démonstration. Supposons que la liste triée des objets $\{a_1, \ldots, a_n\}$ est partitionnée de la façon suivante :

$$A = \{a_i : s_i > 2/3\}, B = \{a_i : 2/3 \geq s_i > 1/2\}, C = \{a_i : 1/2 \geq s_i > 1/3\}, D = \{a_i : 1/3 \geq s_i\}.$$

Soit E' la solution retournée par FirstFitDecreasing. Si au maximum une boîte contient seulement des objets de groupe D, alors il existe une seule boîte (la dernière) remplie à moins de $2/3$, est la borne est prouvée. Dans le cas contraire, on affirme que FirstFitDecreasing trouve une solution optimale. Soit E_0 l'instance obtenu en supprimant de E tous les objets de groupe D. Dans ce cas, les valeurs trouvées par notre heuristique pour E et E_0 sont les mêmes. Donc il suffit de montrer l'optimalité de FirstFitDecreasing pour l'instance E_0. Pour cela, on observe que dans une solution admissible les objets de A sont seuls dans leurs boîtes et que chaque boîte contient au maximum deux objets (donc au maximum un des deux appartient au groupe B). FirstFitDecreasing va essayer de placer chaque objet de C ensemble avec le plus grand objet de B qui ne partage pas sa boîte avec un autre objet. Ça montre que le nombre de boîtes utilisées par FirstFitDecreasing est égal à OPT. □

Une analyse plus détaillée (et plus difficile, puisqu'elle nécessite 70 pages) montre que le nombre de boîtes utilisées par FirstFitDecreasing est $\leq \frac{11}{9}\text{OPT} + 4$.

Chapitre 5

Randomisation

Il existe plusieurs manières d'utiliser l'aléa lors de la conception d'algorithmes. La première, comme dans le cas du tri rapide, consiste à effectuer des actions aléatoirement dans l'objectif de calculer le résultat dans un temps que l'on espère (au sens mathématique) faible. La seconde consiste à calculer avec une complexité garantie un résultat qui est très probablement correct. Ces deux types d'algorithmes portent des noms :

— les algorithmes de Las Vegas : solution garantie, complexité espérée.

— les algorithmes de Monte Carlo : solution probablement correcte, complexité garantie.

Se contenter d'un algorithme probabiliste permet souvent d'obtenir des algorithmes plus performants. Nous allons en étudier quelques exemples.

Avant d'étudier ces applications algorithmiques en détail, on commence par des rappels de probabilités.

5.1 Rappels de probabilités

La définition mathématique de la notion de probabilité se base sur la construction de mondes différents, chacun donnant un des résultats possibles d'une expérience de pensée probabiliste. On note souvent Ω l'univers des mondes possibles. Si on considère le lancer d'un dé à 6 faces, on peut donc considérer un univers $\Omega = \{1, 2, 3, 4, 5, 6\}$ consistant en l'ensemble des tirages possibles. Souvent, on ne s'intéresse pas tant à un élément ω de l'univers Ω en isolation, mais plutôt à une partie de Ω, qu'on appelle *évènement*. Par exemple, « tirer une face paire » est un évènement de l'expérience précédente. Une fois définie un univers, on cherche à évaluer, quantitativement, les résultats qui sont plus ou moins susceptibles de se produire. On associe donc à chaque ω une grandeur numérique.

Dans le cas discret qui va nous intéresser dans ce cours, l'univers Ω est un ensemble fini ou dénombrable (imaginez par exemple le cas où l'univers doit compter le nombre d'opérations élémentaires effectuées par un algorithme ; on a alors $\Omega = \{0, 1, 2, 3, \ldots\} \cup \{+\infty\}$). On associe alors à chaque élément ω sa probabilité $p(\omega) \in [0, 1]$. Pour un évènement A, on lui associe donc une probabilité en sommant les probabilités des évènements élémentaires le composant : $\mathbf{P}(A) = \sum_{\omega \in A} p(\omega)$. Par souci de normalisation, on souhaite avoir toujours $\mathbf{P}(\Omega) = 1$.

Notez que cette définition ensembliste induit de fait que $\mathbf{P}(\emptyset) = 0$ et $\mathbf{P}(A \uplus B) = \mathbf{P}(A) + \mathbf{P}(B)$ si A et B sont deux parties disjointes de Ω. Lorsque A et B ne sont pas disjointes, on a tout de même $\mathbf{P}(A \cup B) \leq \mathbf{P}(A) + \mathbf{P}(B)$, ce qui se généralise à n évènements en :

$$\mathbf{P}(A_1 \cup \cdots \cup A_n) \leq \mathbf{P}(A_1) + \cdots + \mathbf{P}(A_n)$$

Deux évènements A et B sont dits *indépendants* lorsque la connaissance de la réalisation (ou non) de l'un ne modifie pas notre connaissance de la probabilité de l'autre : cela s'écrit formellement $\mathbf{P}(A \cap B) = \mathbf{P}(A) \times \mathbf{P}(B)$.

5.1.1 Accès simultanés à une base de données

Considérons la situation où $n \geq 2$ processus $P_1, P_2, \ldots, P_n$ accèdent de façon compétitive à une base de données. La base peut être utilisée par un seul processus à la fois. On discrétise le temps qui est donc divisé en tours $\{1, 2, 3, 4, \ldots\}$. Pendant un tour, si au moins deux processus essaient d'accéder à la base, alors elle devient verrouillée pendant ce tour. On se demande comment proposer une méthode équitable d'accès pour chaque processus.

Étudions la méthode probabiliste suivante : pour une valeur $0 < p \leq 1$ à déterminer, à chaque tour, chaque processus P_i accède à la base aléatoirement avec probabilité p, de manière indépendante du choix des autres processus.

Considérons les événements suivants :
$A[i,t]$: « P_i essaie d'accéder à la base au tour t »
$S[i,t]$: « P_i réussit à accéder à la base au tour t »
$F[i,t]$: « P_i ne réussit à accéder à la base dans aucun des tours $1, 2, \ldots, t$ »
Sans même avoir à décrire formellement l'univers et les probabilités, on sait que (on note $\overline{A}$ l'évènement complémentaire $\Omega \setminus A$) :

$$\mathbf{P}(A[i,t]) = p, \qquad \mathbf{P}(\overline{A[i,t]}) = 1 - p, \qquad \text{et} \qquad S[i,t] = A[i,t] \cap \left(\bigcap_{j \neq i} \overline{A[j,t]} \right) .$$

Les évènements $A[i,t]$, $\overline{A[j,t]}$ et $\overline{A[k,t]}$ sont indépendants pour i, j et k trois processus distincts (puisque les tirages aléatoires se font de manière indépendante), donc

$$\mathbf{P}(S[i,t]) = \mathbf{P}(A[i,t]) \times \prod_{j \neq i} \mathbf{P}(\overline{A[j,t]}) = p(1-p)^{n-1} .$$

Notons $f(p) = p(1-p)^{n-1}$. Intuitivement, on cherche à maximiser cette probabilité, il est donc naturel d'essayer de choisir p maximisant cette fonction. La dérivée de f vaut $f'(p) = (1-p)^{n-1} - (n-1)p(1-p)^{n-2}$. Elle est donc nulle si et seulement si $p = \frac{1}{n}$, et on peut vérifier qu'il s'agit d'un maximum de la fonction f. Par la suite, posons donc $p = \frac{1}{n}$. Par conséquent, $\mathbf{P}(S[i,t]) = \frac{1}{n}(1 - \frac{1}{n})^{n-1}$.

Puisque $n \geq 2$, on peut voir que la fonction $n \mapsto (1 - \frac{1}{n})^n$ croit de $\frac{1}{4}$ à $\frac{1}{e}$, alors que la fonction $n \mapsto (1 - \frac{1}{n})^{n-1}$ décroit de $\frac{1}{2}$ à $\frac{1}{e}$. Ainsi, $\frac{1}{en} \leq \mathbf{P}(S[i,t]) \leq \frac{1}{2n}$, c'est-à-dire $\mathbf{P}(S[i,t]) = \Theta(\frac{1}{n})$.

On cherche à étudier l'évolution de $\mathbf{P}(F[i,t])$ lorsque t grandit. Or, tous les tours étant indépendants,

$$\mathbf{P}(F[i,t]) = \mathbf{P}\left(\bigcap_{r=1}^{t} \overline{S[i,r]} \right) = \prod_{r=1}^{t} \mathbf{P}(\overline{S[i,r]}) .$$

Puisque $\mathbf{P}(\overline{S[i,r]}) \leq 1 - \frac{1}{en}$, on a

$$\mathbf{P}(F[i,t]) \leq \left(1 - \frac{1}{en}\right)^t .$$

Au tour $t = en$ en particulier,

$$\mathbf{P}(F[i,en]) \leq \left(1 - \frac{1}{en}\right)^{en} \leq \frac{1}{e} .$$

Cela veut dire qu'après $O(n)$ tours, la probabilité d'échec est bornée par une constante $\frac{1}{e}$.

Maintenant, si $t = (en) \cdot (c \log n)$, alors

$$\mathbf{P}(F[i,t]) \leq \left(\left(1 - \frac{1}{en}\right)^{en} \right)^{c \log n} \leq e^{-c \log n} = \frac{1}{n^c} .$$

Ainsi, après $O(n \log n)$ tours, la probabilité d'échec est beaucoup plus petite (qu'après $O(n)$ tours) et bornée par $\frac{1}{n^c}$.

Considérons finalement l'évènement F qu'un des processus n'a pas réussi à accéder la base dans aucune des $t = (en) \cdot (c \log n)$ premiers tours. Si $c = 2$ (c'est-à-dire, $t = 2en \log n$), alors

$$\mathbf{P}(F) = \mathbf{P}\Big(\bigcup_{i=1}^{n} F[i,t]\Big) \leq \sum_{i=1}^{n} \mathbf{P}(F[i,t]) \leq n \cdot \frac{1}{n^2} = \frac{1}{n}\,.$$

Théorème 5.1.1. *Avec une probabilité $\geq 1 - \frac{1}{n}$, les n processus réussissent à accéder à la base au moins une fois après $2en \log n$ tours.*

5.1.2 Variables aléatoires et espérance

Soit $(\Omega, \mathbf{P})$ un espace probabiliste, c'est-à-dire un univers et une fonction de probabilité. Une *variable aléatoire* X est une application $X\colon \Omega \to \mathbb{R}$ tel que pour tout $a \in \mathbb{R}$, $X^{-1}(a) = \{\omega \mid X(\omega) = a\}$ est un évènement. On note $\mathbf{P}[X = a]$ la probabilité $\mathbf{P}(X^{-1}(a))$ de l'évènement $X^{-1}(a)$.

L'*espérance* $\mathbf{E}(X)$ d'une variable aléatoire $X\colon \Omega \to \mathbb{N}$ est définie par

$$\mathbf{E}(X) = \sum_{j=0}^{\infty} j \times \mathbf{P}[X = j]\,.$$

L'espérance est linéaire : si X et Y sont deux variables aléatoires sur Ω, on note $X + Y$ la variable aléatoire égale à $X(\omega) + Y(\omega)$ pour tout $\omega \in \Omega$, et alors, on a $\mathbf{E}(X+Y) = \mathbf{E}(X) + \mathbf{E}(Y)$.

Par exemple, si $\Omega = \{1, \ldots, n\}$ et $\mathbf{P}[X = j] = \frac{1}{n}$ pour tout $j = 1, \ldots, n$, alors

$$\mathbf{E}(X) = 1 \cdot \frac{1}{n} + 2 \cdot \frac{1}{n} + \cdots + n \cdot \frac{1}{n} = \frac{n(n+1)}{2n} = \frac{n+1}{2}\,.$$

Considérons l'expérience consistant au lancer d'une pièce ayant probabilité p de tomber sur *face* et $1 - p$ sur *pile*. On souhaite calculer le nombre moyen (« l'espérance du nombre ») de lancers jusqu'au premier lancer tombant sur *face*. Notons X la variable aléatoire égale au nombre de lancers : l'univers correspond donc à une séquence infinie de lancers de face, c'est-à-dire $\Omega = \{\textit{pile}, \textit{face}\}^{\mathbf{N}}$. Notons que $\mathbf{P}[X = j] = (1-p)^{j-1} \cdot p$: pour qu'on ait besoin d'exactement j lancers, il faut obtenir *pile* aux $j - 1$ premiers lancers et *face* au j-ième lancer, tous les lancers étant indépendants. Donc

$$\mathbf{E}(X) = \sum_{j=0}^{\infty} j \times \mathbf{P}[X = j] = \sum_{j=1}^{\infty} j(1-p)^{j-1}p = \frac{p}{1-p}\sum_{j=1}^{\infty} j(1-p)^{j} = \frac{p}{1-p} \cdot \frac{1-p}{p} = \frac{1}{p}$$

Corollaire 5.1.1. *Si on a une variable de Bernoulli (un pile ou face) avec une probabilité de succès $p > 0$, alors l'espérance du nombre de répétitions jusqu'au premier succès est $\frac{1}{p}$.*

5.2 Tri rapide randomisé

Dans l'algorithme de tri rapide d'un tableau contenant n éléments distincts $S = \{a_1, \ldots, a_n\}$, on choisit un élément a de S comme pivot et on définit les ensembles $S_< = \{a_i \mid a_i < a\}, S_= = \{a_i \mid a_i = a\}$, et $S_> = \{a_i \mid a_i > a\}$. Récursivement, on effectue un appel de l'algorithme sur $S_<$ et/ou $S_>$. On dit que a est un bon pivot si les ensembles $S_<$ et $S_>$ sont relativement petits, c'est-à-dire contiennent au plus une fraction d'éléments par rapport à S (disons au plus $\frac{3}{4}n$ éléments par exemple). Ce pivot peut être choisi de manière aléatoire, auquel cas les algorithmes deviennent eux-même aléatoires : il s'agit alors d'algorithmes de Las Vegas, puisque l'algorithme reste correct quel que soit le choix du pivot. Cependant la complexité de l'algorithme dépend du choix du pivot. Avant toute chose, essayons de comprendre quelle est la probabilité de choisir un *bon pivot*.

Pour répondre à cette question, considérons le tableau trié $[b_1, \ldots, b_n]$ contenant les éléments de $S = \{a_1, \ldots, a_n\}$. Notons que les bons pivots sont exactement les éléments b_i avec $\frac{1}{4}n \leq i \leq \frac{3}{4}n$. Donc parmi les n éléments de S, $\frac{n}{2}$ sont des bons pivots. Autrement dit, pour tout élément $a \in S$, $\mathbf{P}[a \text{ est un bon pivot}] = \frac{1}{2}$.

À la lumière de ce calcul, on peut modifier l'algorithme de tri rapide de façon à vérifier à chaque fois si le pivot courant est un bon pivot et le tirer de nouveau si ce n'est pas le cas. Le résultat précédent et le corollaire 5.1.1 nous permettent de déduire qu'en moyenne cette algorithme tirera le bon pivot deux fois : puisque l'obtention d'un bon pivot à chaque étape induit une complexité en $O(n \log n)$ (utiliser le Master theorem pour vous en convaincre !), on en déduit que cet algorithme modifié a alors une complexité moyenne de l'ordre de $O(n \log n)$.

Mais a-t-on vraiment besoin de cette modification du code ? Une autre façon de faire consiste à considérer qu'on part d'un tableau en entrée du tri rapide qui est lui-même rangé dans un ordre aléatoire. Ayant fixé n éléments distincts $S = \{a_1, \ldots, a_n\}$, il y a $n!$ tableaux différents les contenant. On considère donc une distribution uniforme sur cet ensemble de tableaux différents. On cherche alors la complexité en moyenne du tri rapide, c'est-à-dire l'espérance du nombre de comparaisons effectuées par l'algorithme. Notons $b_1 \leq b_2 \leq \cdots \leq b_n$ les éléments de S trié par ordre croissant. L'univers que l'on considère consiste donc en l'ensemble des permutations des éléments $b_1, b_2, \ldots, b_n$ (qui sont l'ensemble des tableaux possibles en entrée de l'algorithme du tri rapide). Considérons alors la variable aléatoire X_{ij} définie de la façon suivante : $X_{ij}(\omega) = 1$ si les éléments b_i et b_j sont comparés pendant le tri du tableau décrit par la permutation ω, et $X_{ij}(\omega) = 0$ dans le cas contraire. La variable $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$ décrit donc le nombre totale de comparaisons entre deux éléments différents lors de l'exécution du tri rapide. Par additivité de l'espérance, on a

$$\mathbf{E}(X) = \mathbf{E}\left(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{E}(X_{ij})$$

Pour calculer $\mathbf{E}(X)$, il suffit donc de calculer $\mathbf{E}(X_{ij})$.

Notons que

$$\mathbf{E}(X_{ij}) = 1 \cdot \mathbf{P}[X_{ij} = 1] + 0 \cdot \mathbf{P}[X_{ij} = 0] = \mathbf{P}[X_{ij} = 1]\,.$$

Or, $\mathbf{P}[X_{ij} = 1]$ est égale à la probabilité que, au moment où b_i et b_j sont séparés (l'un est inséré dans $S_<$ ou $S_=$ et l'autre est inséré dans $S_=$ ou $S_>$), ces deux éléments seront comparés. Ainsi, le pivot a été nécessairement été choisi parmi $b_i, b_{i+1}, \ldots, b_{j-1}, b_j$ et b_i et b_j sont comparés si le pivot coïncide avec b_i ou b_j. Donc, intuitivement, $\mathbf{P}[X_{ij} = 1] = \frac{2}{j-i+1}$. Par conséquent,

$$\mathbf{E}(X) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{E}(X_{ij}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{P}[X_{ij} = 1] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = 2 \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{1}{k}\,.$$

Puisque que $\sum_{k=1}^{n} \frac{1}{k} = H_n \leq 1 + \ln n$, on obtient $\mathbf{E}(X) \leq 2n(H_n - 1) \leq 2n \ln n$. Par conséquent, l'algorithme de tri rapide a une complexité en moyenne de l'ordre de $O(n \log n)$.

5.3 Algorithme de Karger pour la coupe minimum

Soit $G = (V, E)$ un graphe (non-orienté) et $c\colon E \to \mathbf{N}$ une fonction de capacité. Nous avons vu lors du calcul du flot maximum qu'il est possible de calculer la st-coupe (orientée) d'un graphe (orientée) de capacité minimum d'un graphe. Nous pouvons utiliser cet algorithme pour calculer une plus petite coupe, c'est-à-dire un ensemble de sommets $U \subsetneq V$, $U \neq \emptyset$ et $U \neq V$, qui minimise $\sum_{e \in \delta(U)} c(e)$, avec $\delta(U) = \{e = (u, v) \mid u \in U, v \notin U\}$.

Pour cela, il suffit de remplacer chaque arête non orientée e par deux arcs entre la même paire de sommets, dans chaque sens, ayant la même capacité $c(e)$. Ensuite, il faut tester toutes les paires (s, t) possibles et garder celles ayant la plus petite coupe. Cet algorithme, en utilisant

l'algorithme de Ford-Fulkerson avec calcul des plus courts chemins, aurait une complexité asymptotique $O(n^3m^2)$. Nous allons utiliser une source d'aléa pour calculer plus rapidement une coupe qui est minimum avec une probabilité $\Omega(1/n^2)$.

Nous commençons par montrer que les arêtes d'une coupe minimum sont rares dans le graphe.

Lemme 5.3.1. *Notons $C = \sum_{e\in E} c(e)$. Soit k la capacité minimum d'une coupe de G. Alors $C \geq nk/2$ (c'est-à-dire $k \leq 2C/n$).*

Démonstration. Pour tout sommet u, la coupe $\delta(u)$ (ou $\delta(\{u\})$) a une capacité au moins égale à la coupe minimum. En sommant sur tous les sommets, on obtient :

$$\sum_{u\in V}\sum_{e\in\delta(u)} c(e) \geq nk$$

Par ailleurs, dans cette sommation, chaque arête est comptée deux fois, une fois à chacune de ses extrémités. Ce qui donne :

$$\sum_{u\in V}\sum_{e\in\delta(u)} c(e) = 2\sum_{e\in E} c_e = 2C$$

Donc $2C \geq nk$, c'est-à-dire $C \geq nk/2$ □

Ainsi :

Lemme 5.3.2. *Soit U une coupe minimum de G. Soit e une arête choisie aléatoirement avec une distribution proportionnelle aux capacités des arcs (c'est-à-dire $\mathbf{P}(e \text{ est choisie}) = c_e/C$). Alors la probabilité que e soit dans la coupe minimum considérée est*

$$\mathbf{P}(e \in \delta(U)) \leq \frac{2}{n}\,.$$

Démonstration. En utilisant le lemme précédent, il vient

$$\mathbf{P}(e \in \delta(u)) = \sum_{e\in\delta(u)} \frac{c_e}{C} = \frac{k}{C} \leq \frac{2}{n}\,.$$

□

Nous pouvons donc choisir une arête aléatoirement et avoir une bonne probabilité qu'elle ne fasse pas partie de la coupe minimum U. Notez qu'il peut y avoir plusieurs coupes minimums, et que possiblement toutes les arêtes sont dans au moins une coupe minimum. Il est donc important ici que U soit une coupe minimum fixée, et ce que nous allons trouver, c'est un algorithme qui trouve cette coupe U avec une probabilité $\Omega(1/n^2)$.

Une fois choisie une arête qui ne sera pas dans la coupe, nous pouvons identifier ses deux extrémités en un seul sommet.

Définition 5.3.1. *Soit $G = (V, E)$ un multi-graphe (c'est-à-dire un graphe orienté dans lequel on a le droit d'avoir plusieurs arcs ayant la même source et la même destination) et $e = (u, v) \in E$. Le graphe G/e,* contraction *du graphe G par l'arête e, est le multi-graphe (V', E') défini par :*

- *$V' = V \setminus \{v\}$,*
- *$E' = E \setminus \{e\}$,*
- *$dst_{G'}(e') = dst_G(e')$ si $dst_G(e') \neq v$, $dst_{G'}(e') = u$ sinon,*
- *$src_{G'}(e') = src_G(e')$ si $src_G(e') \neq v$, $src_{G'}(e') = u$ sinon.*

Ainsi, seul l'arc $e = (u, v)$ disparait, les autres arcs incidents à v sont incidents à u dans G/e. Notons que toute coupe $\delta(U')$ dans G/e est aussi une coupe $\delta(U)$ dans G avec $U = U'$ si $u \notin U'$ et $U = U' \cup \{v\}$ si $u \in U'$. Donc la capacité minimale d'une coupe de G' est au moins celle d'une coupe de G.

Notre algorithme est le suivant : choisir une arête aléatoirement, la contracter, et répéter ces deux opérations jusqu'à avoir un graphe à deux sommets. Un graphe à deux sommets possède une seule coupe. Nous bornons la probabilité que l'algorithme choisisse une arête de U. Chaque itération, s'il reste i sommets, il ne se trompe pas avec une probabilité au moins $1 - 2/i$. La probabilité qu'il se trompe jamais est donc au moins :

$$\sum_{i=3}^{n} \left(1 - \frac{2}{i}\right) = \sum_{i=1}^{n-2} \frac{i}{i+2} = \frac{2}{n(n-1)}$$

Ainsi, en répétant l'algorithme $T = \frac{n(n-1)\ln n}{2}$ fois, une des coupes de plus petite capacité trouvées est très probablement U : en effet, la probabilité qu'aucune ne soit une coupe minimum vaut

$$\left(1 - \frac{2}{n(n-1)}\right)^{T} \leq \left(\frac{1}{e}\right)^{\ln n} = \frac{1}{n}$$

la première inégalité provenant du fait que $(1-x)^{1/x} \leq 1/e$ pour tout $x \in]0,1[$.

C'est un exemple d'algorithme de Monte Carlo qui a une complexité garantie, et fournit un résultat correct avec une grande probabilité.