Algorithmique II

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*, ...) : *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 algo-

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

FIGURE 1 – La structure de données abstraite *liste*

rithmes plus complexes, en faisant abstraction de leur implémentation. Nous faisons donc la distinction entre structures de données *abstraite* et *concrète*.

Définition 1.1. *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. *Une* structure de données concrète *(ou* structure de donnée*) est une description de l'implantation d'une structure de donnée abstraite dans un modèle de calcul précis. Dans le cadre de ce cours, cette description est un pseudocode pour le modèle RAM.*

Exemple 1.1 (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 persistentes : les fonctions d'insertion et du suppression ne modifie pas la liste passée en argument mais retourne une liste fraîche.

La Figure 1 donne la spécification de cette structure de donnée 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 (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 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 persistentes : 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.

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 2 – La structure de données abstraite *pile*

Exemple 1.3 (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 3.

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

— le type *maillon* est défini comme une disjonction (**ou bien**). Certains langages 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 requiert 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 2 et Figure 1. En fait, elles peuvent être vues comme des variantes l'un 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 persistente : 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 persistentes ont cet avantage de ne jamais être modifiées ou détruites qui peut être utile pour écrire des algorithmes. En échange, la persistence peut avoir un coût sur l'utilisation de la mémoire, et dans certains cas les meilleurs algorithmes persistents sont moins efficaces que leurs contreparties

```
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 3 – Pseudocode pour les piles codées par des listes simplement chainées

destructives.

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

Exemple 1.4 (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 donnée 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 donné par la Figure 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 4). Un exemple de représentation de tas est donné en Figure 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 7. Pour rester simple, nous supposons que nous disposons d'une implémentation de la structure de donnée abstraite *tableau dynamique*, qui sont des tableaux dont la taille est variable et augmente automatiquement en fonction des besoins.

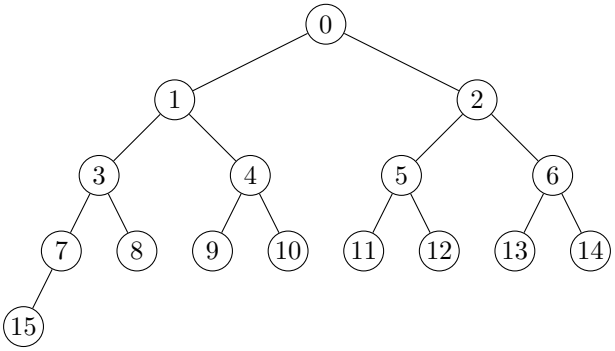
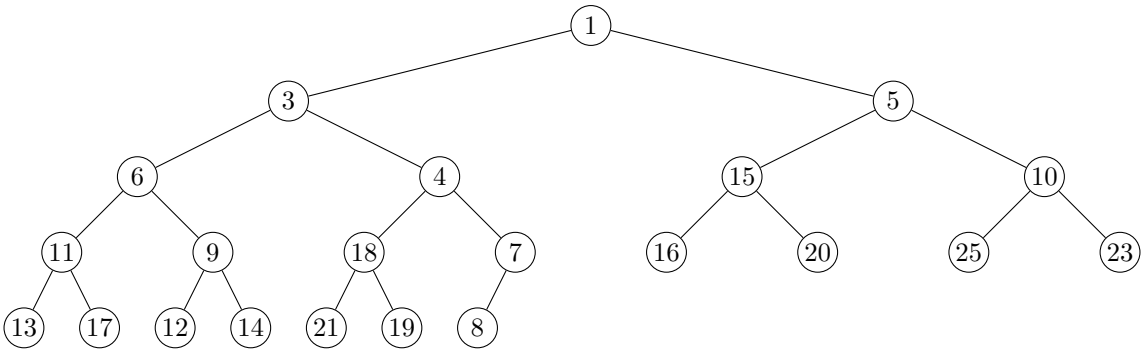


FIGURE 4 – Indexation des nœuds d'un tas binaire pour la représentation par tableau.



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

FIGURE 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 6 – La structure de données abstraite *file de priorité* (variante destructive)

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 donnée 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. *Un* invariant d'un type de donnée *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 1.4. *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 1.5. *Une* précondition *d'un algorithme est une propriété sur les données de l'algorithme, nécessaire à la correction de cet algorithme.*

Définition 1.6. *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.*

```
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 7 – La structure de données concrète *tas binaire*

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 1.5 (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'interprêtent sémantiquement comme des multi-ensembles d'éléments ordonnés (*cf.* Figure 6). Les invariants sont :

$$[\![\mathsf{longueur}(\mathsf{h})]\!] = |H| \tag{1}$$

$$H = \{[\![\mathsf{h}.[i]]\!] \ : \ i \in [0, |H| - 1]\} \tag{2}$$

$$\text{pour tout } i \in [1, |H| - 1], [\![\mathsf{h}.[\mathsf{pere}(i)]]\!] \leq [\![\mathsf{h}.[i]]\!] \tag{3}$$

(1) assure que le tableau est de même taille que le tas, (2) certifie que ce tableau contient bien tous les éléments du tas précisement. (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 $\mathsf{insère}$ de la Figure 7. Pour cela, nous aurons besoin d'une précondition pour la fonction $\mathsf{remonte}$, qui est une relaxation de l'invariant (3). On définit comme précondition de $\mathsf{remonte}(\mathsf{h}, \mathsf{noeud})$:

$$\text{pour tout } i \in [1, |H| - 1] \setminus [\![\mathsf{noeud}]\!], [\![\mathsf{h}.[\mathsf{pere}(i)]]\!] \leq [\![\mathsf{h}.[i]]\!] \tag{4}$$

Démonstration. Prouvons d'abord la fonction $\mathsf{remonte}$, en supposant les invariants de types (1) et (2) et la précondition (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 $\mathsf{remonte}$. Nous procédons par récurrence sur $[\![\mathsf{noeud}]\!]$.

Si $[\![\mathsf{noeud}]\!] = 0$, (4) s'instancie exactement en (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 $\mathsf{remonte}$.

Si $i := [\![\mathsf{noeud}]\!] \neq 0$ et que avant l'exécution de $\mathsf{remonte}$, $[\![\mathsf{h}.[i]]\!] > [\![\mathsf{h}.(\mathsf{pere}(i))]\!]$, alors par la précondition (4), l'invariant (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) et (2), puisqu'on échange seulement la position de deux éléments. Par la précondition (4), comme seul les nœuds d'indice i et $[\![\mathsf{pere}(i)]\!]$ sont modifiés, la précondition (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 $\mathsf{remonte}$.

Ceci termine la preuve de correction de $\mathsf{remonte}$. La preuve de correction de $\mathsf{insère}$ 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 $\mathsf{insère}$, telle que définit par la structure de données abstraire 6, est respectée.

L'insertion d'un élément à l'indice l garantie que la longueur du tableau augmente de 1, donc (1) est satisfait avant l'appel à la fonction $\mathsf{remonte}$. Puisque l'élément inséré est elt, et selon la sémantique de l'opération d'insertion dans les tas, (2) est aussi vrai. Enfin, tout noeud présent dans h au début de l'exécution de $\mathsf{insère}$ garde le même contenu ainsi que son père, donc (3) ne peut être violée que par le nouveau nœud d'indice l, donc la précondition (4) est vérifié pour l'appel de $\mathsf{remonte}$ 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 1.7. *Soit $f : \mathbb{N}^* \to \mathbb{N}^*$, $g : \mathbb{N}^* \to \mathbb{N}^*$ deux fonctions, on dit que :*

- *f est négligeable devant g si pour tout $\epsilon > 0$, il existe $N \in \mathbb{N}^*$ tel que pour tout $n \geq N$, $f(n) \leq \epsilon \cdot g(n)$, 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 $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$, et on le note $f \sim g$.*

Exemple 1.6 (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 $k * (p + 1)$, 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 8), avec les structures concrètes par liste simplement ou doublement chainées et par tableau circulaire,

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

FIGURE 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*	$[\![\text{dictionnaire\_vide}()]\!] = \varnothing$
est_vide(*dictionnaire*) : *bool*	$[\![\text{est\_vide(dico)}]\!] =$ vrai si $[\![\text{dico}]\!] = \varnothing$, $[\![\text{est\_vide(dico)}]\!] =$ faux sinon
insère(t, *dictionnaire*) : *dictionnaire*	$[\![\text{insère(elt, dico)}]\!] = \{\text{elt}\} \cup [\![\text{dico}]\!]$
appartient(t, *dictionnaire*) : *bool*	$[\![\text{appartient(elt, dico)}]\!] = [\![\text{elt}]\!] \in [\![\text{dico}]\!]$
supprime(t, *dictionnaire*) : *dictionnaire*	$[\![\text{supprime(elt, dico)}]\!] = [\![\text{dico}]\!] \setminus \{\text{elt}\}$,

FIGURE 9 – La structure de données abstraite *dictionnaire* (variante persistante)

	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 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,

2 Structures de donnée

2.1 Équilibrage des arbres binaires de recherche

2.1.1 Considérations générales

Les arbres binaires de recherche forment une famille de structure de données concrêtes implémentant la structure abstraite de dictionnaire, présentée en Figure 9. Leur principal but est donc de mémoriser un ensemble d'éléments afin de pourvoir tester l'appartenance d'un élément quelconque à cet ensemble. Les autres propriétés les plus désirables sont l'ajout et la suppression d'éléments dans l'ensemble.

Les arbres binaires de recherches, correctement implantés, permettent d'avoir des complexités de $O(\log n)$ pour ces trois opérations, où n est le nombre d'éléments de l'ensemble (ce qui est optimal si nous nous en tenons aux structures persistentes, nous verrons plus tard comment faire mieux en destructif).

Définition 2.1. *Un* arbre binaire T *sur un ensemble de* nœuds N *est ou bien l'*arbre vide $\perp$ *si* $N = \varnothing$*, ou bien un triplet* (n, T_g, T_d)*, avec* $n \in N$*,* T_g *un arbre binaire sur* N_g *et* T_d *un arbre binaire sur* N_d*, avec* $N_g \uplus N_d = N \setminus \{n\}$*.*

Dans le deuxième cas, racine$(T) = n$ *est la* racine *de* T*,* T_g *est le* sous-arbre gauche *de* n*,* T_d *est le* sous-arbre droit *de* n*. Si* $T_g \neq \perp$*, enfant_gauche*$(n) =$ *racine*(T_g) *est le* fils gauche *de* n*, si* $T_d \neq \perp$*, enfant_droit*$(n) =$ *racine*(T_d) *est le* fils droit *de* n*. Le* père *parent*(n') *d'un noeud* n' *est l'unique nœud* n *tel que* $n' =$ *enfant_gauche*(n) *ou* $n' =$ *enfant_droit*(n)*. Nous dénotons* $N(T) = N$ *l'ensemble des nœuds de l'arbre* T*.*

La hauteur *d'un arbre, notée* h*, est définie par* $h(\perp) = 0$*,* $h(n, T_1, T_2) = 1 + \max\{h(T_1), h(T_2)\}$*. La* profondeur *d'un nœud* n *dans l'arbre* T *est définie comme* $h(T) - h(T_n)$*, où* T_n *est le sous-arbre de* T *de racine* n*. En particulier, la racine d'un arbre* T *est l'unique nœud de profondeur* 0 *dans* T*.*

Nous représenterons les arbres binaires sous la forme classique : l'arbre est dessiné récursivement en plaçant la racine en haut, les sous-arbres gauche et droit en dessous respectivement à gauche et à droite. Chaque nœud est lié par une arête à son père, comme dans la Figure 10.

Les arbres binaires de recherche imposent une condition supplémentaire sur l'ordre des nœuds dans la structure de l'arbre. L'idée primordiale est de comparer l'élément recherchée avec l'élément en racine. La meilleure façon d'exploiter le résultat de la comparaison serait de réduire de moitié l'espace de recherche. Étant donné la structure d'un arbre binaire, le plus simple serait de pouvoir disqualifier l'un des deux sous-arbres de la racine. Pour cela, nous allons supposer que le sous-arbre droit contient uniquement des éléments plus petits que la racine, et le sous-arbre gauche des éléments plus grand. Ainsi après la comparaison avec la racine il suffit de chercher dans un seul des deux sous-arbres. Si ceux-ci sont de même taille, cela réduit bien l'espace de recherche par deux, ce qui nous garantirait un temps de recherche logarithmique par rapport à la taille de l'ensemble encodé.

Nous définissons donc les arbres binaires de recherche :

Définition 2.2. *Un arbre binaire de recherche est un arbre binaire sur un ensemble ordonné de nœuds (par un ordre* $\leq$*), tel que pour tout sous-arbre* (n, T_g, T_d) *:*

$$(\forall n_g \in N(T_g), n_g < n) \wedge (\forall n_d \in N(T_d), n < n_d) \tag{5}$$

(tout nœud du sous-arbre gauche de n *est plus petit que* n*, tout nœud du sous-arbre droit de* n *est plus grand que* n*).*

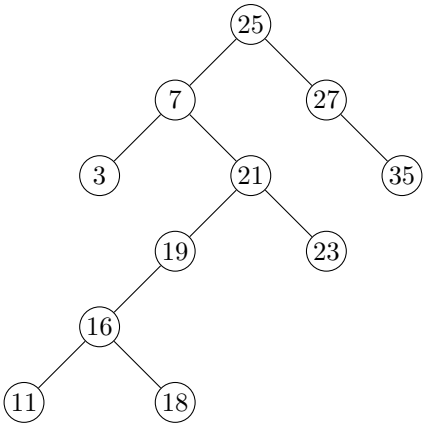


FIGURE 10 – Un arbre binaire de recherche quelconque

Par abus de notations, nous écrirons $S_1 < e$ et $e < S_2$ si respectivement $\forall s \in S_1, s_1 < e$ et $\forall s \in S_2, e < s_2$. Ainsi, la condition (5) se réécrit simplement $N(T_g) < n < N(T_d)$.

Exemple 2.1. La Figure 10 représente un arbre binaire de recherche sur l'ensemble de nœuds

$$\{3; 7; 11; 16; 18; 19; 21; 23; 25; 27; 35\}.$$

Cet arbre a une hauteur de 6, a pour racine l'élément 25. Le sous-arbre gauche de la racine contient 8 éléments, le sous-arbre droit en contient 2.

Nous nous inspirons de la définition inductive des arbres binaires de recherche, pour proposer une implantation inductive du type des arbres binaires, en notant $\perp$ l'arbre vide :

```
1  type arbre_binaire de t =
2    struct{
3      sous_arbre_gauche : arbre_binaire;
4      contenu : t;
5      sous_arbre_droit : arbre_binaire;
6    } ou bien ⊥
```

Nous pouvons alors exprimer les propriétés des arbres binaires, en particulier la sémantique d'un arbre binaire :

$$[\![\mathsf{arbre}]\!] := \begin{cases} \varnothing & \text{si } \mathsf{arbre} = \perp \\ [\![\mathsf{arbre.sous\_arbre\_droit}]\!] \cup \{[\![\mathsf{arbre.contenu}]\!]\} \cup [\![\mathsf{arbre.sous\_arbre\_gauche}]\!] & \text{sinon} \end{cases}$$

ce qui exprime que le dictionnaire encodé par un arbre binaire est l'ensemble de tous ses nœuds, le nœud racine et les nœuds de ses deux sous-arbres.

Les invariants du type *arbre_binaire* doivent contenir ceux des arbres binaires de recherche. Pour un arbre

```
1  fonction appartient(t elt, arbre_binaire t) : bool =
2    si t ≠ ⊥ alors retourner faux
3    si elt = t.contenu alors retourner vrai
4    si t.contenu ≤ elt alors
5      retourner appartient(elt, t.sous_arbre_droit)
6    sinon
7      retourner appartient(elt, t.sous_arbre_gauche)))
```

FIGURE 11 – Test d'appartenance dans un arbre binaire de recherche

binaire t codant un ensemble S :

$$[\![\mathsf{t}]\!] = S \tag{6}$$

$$\mathsf{t} \neq \bot \Rightarrow [\![\mathsf{t.sous\_arbre\_gauche}]\!] < [\![\mathsf{t.contenu}]\!] \tag{7}$$

$$\mathsf{t} \neq \bot \Rightarrow [\![\mathsf{t.contenu}]\!] < [\![\mathsf{t.sous\_arbre\_droit}]\!] \tag{8}$$

Nous reprenons l'idée à la base des arbres binaires de recherche pour coder l'opération appartient, consistant à tester l'appartenance d'un élément à un ensemble. Comme nous l'avions annoncé, il suffit de comparer l'élément recherché à la racine, puis de faire un appel récursif sur l'un de deux sous-arbres. Le pseudocode est donné par la Figure 11.

Correction : Supposons que t est un arbre binaire de recherche valide. Les lignes 2 et 3 traitent des cas lorsque l'arbre est vide, ou la racine est l'élément cherché, et sont d'évidence correctes. Si ces deux premiers tests échouent, les lignes 5 et 6 font un appel récursif dans un des deux sous-arbres. Pour prouver que l'algorithme est correct, il suffit de montrer que l'autre sous-arbre ne contient pas l'élément recherché. Si t.contenu $<$ elt, par l'invariant de type (7), $[\![\mathsf{t.sous\_arbre\_gauche}]\!] < [\![\mathsf{elt}]\!]$, donc elt n'est pas dans le sous-arbre gauche. Inversement si elt $<$ t.contenu, par l'invariant de type (8), $[\![\mathsf{elt}]\!] < [\![\mathsf{t.sous\_arbre\_droit}]\!]$, donc elt n'est pas dans le sous-arbre droit. L'algorithme termine par induction sur la structure de l'arbre, donc il est correct.

Complexité : Plus précisement, le nombre d'appel récursif est égal à la profondeur du nœud contenant l'élément recherché elt s'il est présent dans l'arbre, ou 1 plus la profondeur d'un autre nœud sinon (précisement, le plus petit nœud plus grand que elt ou le plus grand nœud plus petit que elt, mais nous ne le démontrons pas). Dans le pire des cas, la profondeur d'un nœud peut être $n-1$, comme sur la Figure 12. Hormis les appels récursifs, les autres opérations sont élémentaires, on en déduit une complexité dans le pire des cas en $O(h(T))$ pour l'appartenance à un arbre T.

Nous en déduisons que pour obtenir une bonne garantie de performance sur nos arbres binaires de recherche, il faut nous assurer que la hauteur de l'arbre reste contenue. Toute une littérature existe sur les techniques pour assurer un bon équilibre des arbres binaires de recherche. Nous abordons une de ces techniques dans la prochaine section.

2.1.2 Définition des arbres AVL

La méthode la plus immédiate pour s'assurer qu'un arbre est bien équilibré, est de spécifier que les deux sous-arbres d'un nœud quelconque sont approximativement de même cardinalité. Les AVL-arbres, du nom

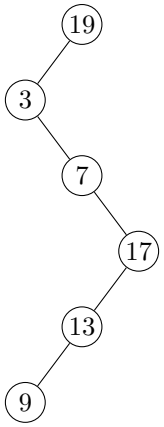


FIGURE 12 – Un arbre binaire de recherche extrêmement déséquilibré, le nœud 9 a une profondeur de 5, alors qu'il n'y a que 6 nœuds en tout.

des découvreurs Adelson-Velskii et Landis, relâche cette contrainte à être de même hauteur, à 1 près. Nous prouverons qu'ainsi la hauteur d'un AVL-arbre est toujours au plus logarithmique en le nombre de nœuds.

Définition 2.3. *Un* AVL-arbre *est un arbre binaire de recherche T qui vérifie la propriété :*

$$\textit{pour tout sous-arbre } (n, T_g, T_d) \textit{ de } T,\ |h(T_g) - h(T_d)| \leq 1 \tag{9}$$

Lemme 2.1. *Soit T un AVL-arbre à n nœud, $h(T) \leq 1.45 \log_2 n$.*

Démonstration. Notons A_h un AVL-arbre de hauteur h ayant un nombre minimum de nœuds, noté n_h. A_0 est donc l'arbre avec un seul nœud, la racine, donc $n_0 = 1$. A_1 est un arbre de hauteur 1, avec une racine ayant un seul fils.

A_n a pour hauteur n, donc l'un des sous-arbres de sa racine a pour hauteur $n-1$, par symétrie on peut supposer que c'est le sous-arbre droit. Par la définition des AVL-arbres (9), le sous-arbre gauche doit avoir pour hauteur $n-2$ ou $n-1$. Par minimalité du nombre de nœuds, chaque sous-arbre de la racine minimise le nombre de nœud relativement à sa hauteur, donc $A_n = (\text{racine}(A_n), A_{n-2}, A_{n-1})$ (voir Figure 13). Nous en déduisons la relation de récurrence :

$$n_h = 1 + n_{h-2} + n_{h-1}$$

avec $n_0 = 1$, $n_1 = 2$. Posons $p_h := n_h + 1$ pour tout h, nous obtenons :

$$p_h = p_{h-1} + p_{h-2}$$

avec $p_0 = 2$, $p_1 = 3$. Pour analyser les récurrences de la forme $p_n = a \cdot p_{n-1} + b \cdot p_{n-2}$, il suffit de trouver les racines r_1 et r_2 du polynôme caractéristique correspondant $x^2 - ax - b$, et alors $p_n = \alpha \cdot r_1^n + \beta \cdot r_2^n$. Les valeurs de α et β sont déduites des conditions initiales. Dans notre cas, les racines de $x^2 - x - 1$ sont $\frac{1+\sqrt{5}}{2}$ et $\frac{1-\sqrt{5}}{2}$, donc :

$$p_h = \alpha \left(\frac{1+\sqrt{5}}{2}\right)^h + \beta \left(\frac{1-\sqrt{5}}{2}\right)^h$$

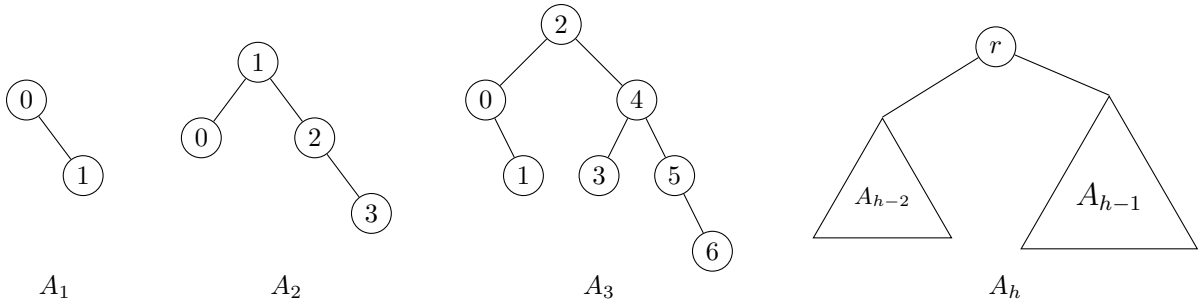


FIGURE 13 – Les arbres de Fibonacci, les plus déséquilibrés des AVL-arbres.

Par les conditions initiales, $\beta = 2 - \alpha$ et $\alpha \cdot \sqrt{5} = 2 + \sqrt{5}$, donc

$$n_h = p_h - 1 = \left(1 + \frac{2}{\sqrt{5}}\right) \cdot \left(\frac{1+\sqrt{5}}{2}\right)^h + \left(1 - \frac{2}{\sqrt{5}}\right) \cdot \left(\frac{1-\sqrt{5}}{2}\right)^h - 1$$

Le second terme de cette expression tend rapidement vers 0 et sa valeur absolue reste inférieur à 1, donc

$$n_h \geq \left(1 + \frac{2}{\sqrt{5}}\right) \cdot \left(\frac{1+\sqrt{5}}{2}\right)^h$$

Par passage au logarithme :

$$\log_2 n_h \geq h \log_2 \left(\frac{1+\sqrt{5}}{2}\right)$$

et en évaluant nous obtenons $h \leq 1.45 \log_2 n_h$. □

Nous devons maintenant trouver comment mettre en œuvre ce principe de façon algorithmique. Les principales difficultés vont être de réaliser l'insertion et la suppression d'éléments d'un arbre sans violer la condition d'équilibrage (9). Le test d'appartenance n'est en revanche pas modifié, nous utilisons simplement l'algorithme 11, qui sur les AVL-arbres, au regard de l'analyse que nous avions donné, a une complexité dans le pire des cas de $O(\log_2 n)$ pour un arbre à n nœuds.

Nous commençons la construction des algorithmes d'insertion et suppression en renforçant le type de données des arbres binaires de recherche en ajoutant à chaque nœud la hauteur du sous-arbre enraciné en ce nœud.

```
1  type avl_arbre de t =
2    struct{
3      sous_arbre_gauche : arbre_binaire;
4      contenu : t;
5      sous_arbre_droit : arbre_binaire;
6      hauteur : entier;
7    } ou bien ⊥
```

Rappelons que la sémantique d'un AVL-arbre est définie comme l'ensemble de ses nœuds, défini récursivement par $[\![\bot]\!] = \varnothing$ et

$$[\![\text{arbre}]\!] = \{[\![\text{arbre.contenu}]\!]\} \cup [\![\text{arbre.sous\_arbre\_gauche}]\!] \cup [\![\text{arbre.sous\_arbre\_droit}]\!]$$

Nous pouvons maintenant exprimer les invariants que nous souhaitons. Pour un arbre $\mathsf{arbre} \neq \bot$ de type *avl_arbre*, codant un ensemble S d'éléments (par convention nous posons $\bot.\mathsf{hauteur} = -1$, ce qui est un abus de notation car $\bot$ n'a pas de champs hauteur), les invariants de type sont :

$$[\![\mathsf{arbre}]\!] = S \tag{10}$$

$$[\![\mathsf{arbre.sous\_arbre\_gauche}]\!] \leq [\![\mathsf{arbre.contenu}]\!] \leq [\![\mathsf{arbre.sous\_arbre\_droit}]\!] \tag{11}$$

$$\mathsf{arbre.hauteur} = 1 + \max\{\mathsf{arbre.sous\_arbre\_gauche.hauteur}; \mathsf{arbre.sous\_arbre\_droit.hauteur}\} \tag{12}$$

$$|\mathsf{arbre.sous\_arbre\_gauche.hauteur} - \mathsf{arbre.sous\_arbre\_droit.hauteur}| \leq 1 \tag{13}$$

La condition (11) exprime que les AVL-arbres sont des arbres binaires de recherche. La condition (12) définit la hauteur d'un nœud. La condition (13) est la contrainte propre aux AVL-arbres, telle que dans la définition de ceux-ci.

2.1.3 Insertion dans un arbre AVL

L'insertion dans un arbre binaire de recherche consiste à descendre dans l'arbre comme dans un test d'appartenance, jusqu'à la feuille du dernier appel récursif : on remplace alors cette feuille par un nouveau nœud contenant l'élément à insérer. Cette algorithme permet de maintenir l'invariant de type des arbres binaires de recherche. Nous allons nous baser sur cet algorithme pour concevoir l'insertion dans les AVL-arbres. Nous allons avoir deux problèmes supplémentaires :

— le premier facile : il faut recalculer les hauteurs des sous-arbres (invariant de type (12)),

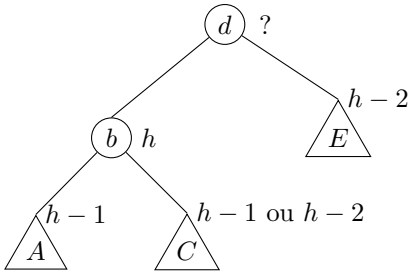
— le second difficile : il faut préserver la propriété spécifique des AVL concernant l'équilibrage.

Pour traiter le second problème, l'intuition primordiale tient dans le Lemme 2.1 : le nombre de nœuds d'un sous-arbre a une très forte dépendance en sa hauteur. Autrement dit, en général plus un arbre est haut, plus il possède de nœuds. Cela peut-être faux parfois (un arbre complet de hauteur n a plus de nœuds qu'un AVL-arbre minimum de hauteur $n-1$ par exemple), mais il s'agit néanmoins d'une bonne approximation, qui guide les choix algorithmiques de cette section.

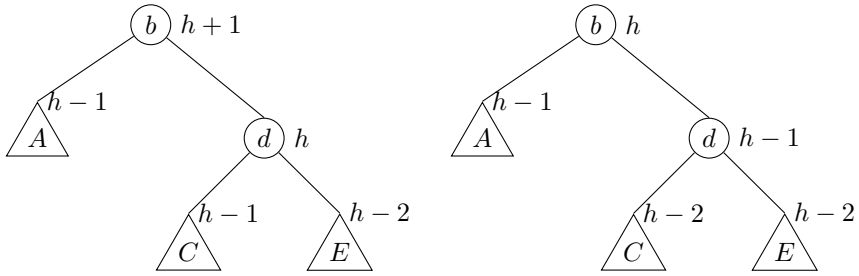
À la suite de l'insertion d'un élément, tout sous-arbre dans lequel a été effectué l'insertion peut voir sa hauteur augmentée d'au plus 1. Tout nœud d'un tel sous-arbre peut donc se retrouver à violer la condition d'équilibre. Précisement, les situations de violation sont décrites par cette figure :

Nous ne considérons que la situation de gauche. Nous allons distinguer deux cas, selon les hauteurs des sous-arbres du sous-arbre gauche. Le sous-arbre gauche A a une hauteur h excédant de 2 la hauteur du sous-arbre droit C. Intuitivement, il faut comprendre que A possède trop de nœuds comparé à C : nous voudrions donc transférer une partie des nœuds de A vers C. Dans cette optique, nous devons examiner la structure de A (qui est garantie d'être non-vide puisque de hauteur au moins 1). Nous distinguons à nouveau deux cas.

Cas 1 :

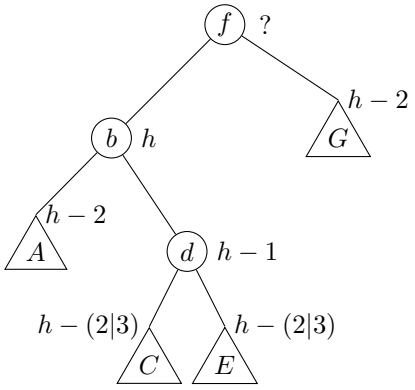


Le sous-arbre gauche est trop gros, nous déplaçons donc une partie du sous-arbre gauche vers le sous-arbre droit. Une façon de faire et de mettre C, d, E dans le sous-arbre droit. b devient alors la racine :



Dans les deux cas, l'invariant d'équilibrage (13) est localement rétablie (mais peut maintenant être faux au niveau du père de b dans le premier cas).

Cas 2 :



Cette fois encore le sous-arbre gauche est trop gros, mais A n'est pas assez gros pour équilibrer la somme des autres sous-arbres. Nous sommes donc obligé de couper le sous-arbre droit de b en deux (c'est possible car sa hauteur est $h - 1 \geq 1$)pour trouver une répartition plus équitable. Nous pouvons alors placer A, b, C à gauche, d en racine, le reste à droite.

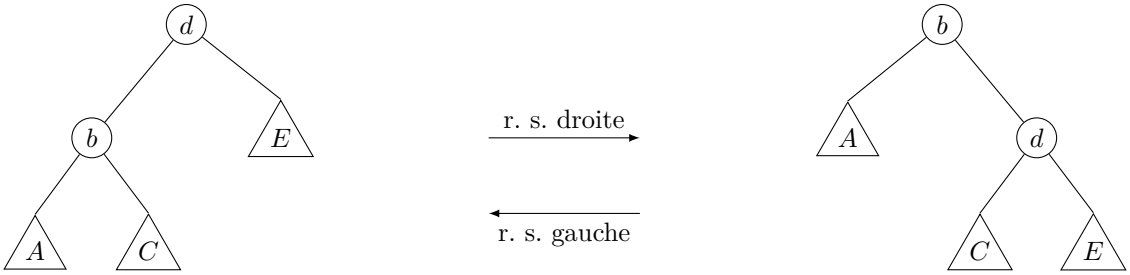
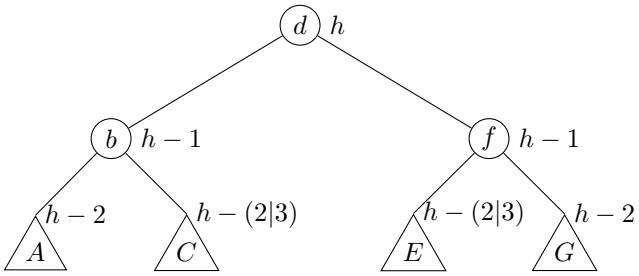


FIGURE 14 – Rotations simples



Nous venons donc d'exhiber deux opérations locales sur des arbres binaires de recherche qui devraient suffire pour garantir les invariants des AVL-arbres. Commençons par nommer ces deux opérations.

Définition 2.4. *La* rotation simple droite *d'un arbre binaire de recherche* $(d, (b, A, C), E)$ *est l'arbre binaire de recherce* $(b, A, (d, C, E))$. *La* rotation simple gauche *est l'opération inverse. Ces opérations sont illustrés en Figure 14.*

Définition 2.5. *La* rotation double droite *d'un arbre binaire de recherche* $(f, (b, A, (d, C, E)), G)$ *est l'arbre binaire de recherche* $(d, (A, b, C), (E, f, G))$. *La* rotation double gauche *d'un arbre binaire de recherche* $(b, A, (f, (C, d, E), G))$ *est l'arbre binaire de recherche* $(d, (A, b, C), (E, f, G))$ *(Figure 15).*

Dans les deux définitions, il est nécessaire de montrer que les invariants d'arbres binaires de recherche sont préservés. Clairement l'ensemble des nœuds est préservé (il n'y a ni nouveau nœud, ni nœud qui disparait), il suffit donc de montrer la condition (5) sur l'ordre des noeuds. Ici, le choix de notation lexicographique des nœuds et des sous-arbres permet de le vérifier d'un regard : les nœuds et sous-arbres sont bien triés par ordre infixe avant et après rotation.

Nous allons maintenant mettre en œuvre ces deux opérations pour coder l'insertion dans un AVL-arbre. La première étape est de construire deux fonctions de construction d'arbre qui permettent de garantir la condition d'équilibrage (13). Chacune de ces deux fonctions prend deux AVL-arbres G et D et un nœud n avec $G < c < D$, et construit un AVL-arbre équivalent à (n, G, D). La différence entre les deux tient dans la précondition :

- joint_gauche(*avl_arbre* $G, n,$ *avl_arbre* D) : *avl_arbre*.
  Précondition : $h(G) - h(D) \in \{-1, 0, 1, 2\}$ (s'il y a déséquilibre, c'est G qui est trop haut),

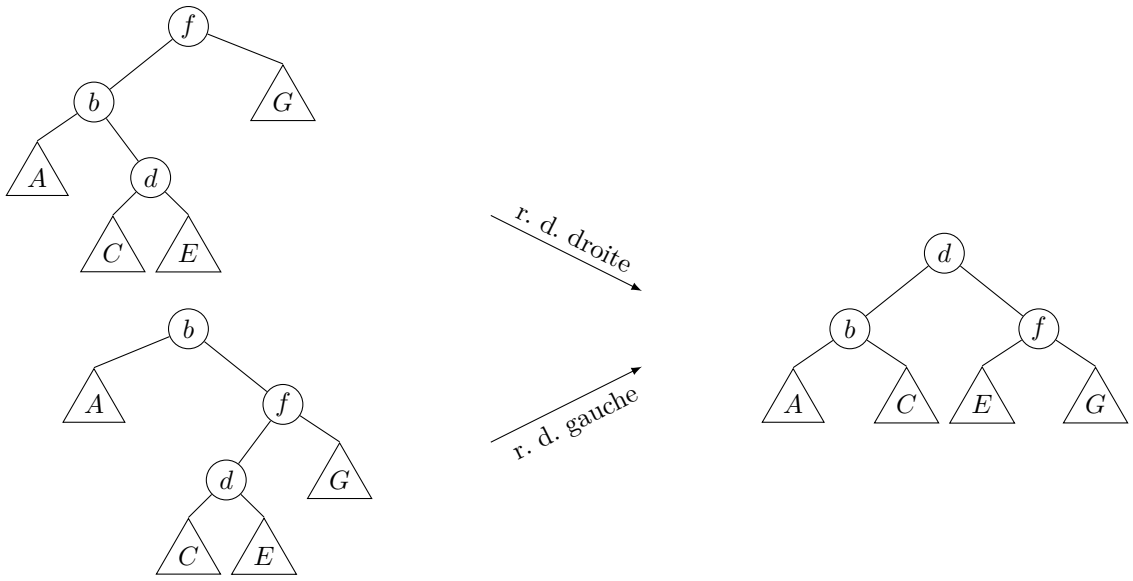


FIGURE 15 – Rotations doubles

- joint_droit(*avl_arbre* $G, n,$ *avl_arbre* D) : *avl_arbre*.
  Précondition : $h(D) - h(G) \in \{-1, 0, 1, 2\}$ (s'il y a déséquilibre, c'est D qui est trop haut),

joint_gauche sera utilisé après l'insertion dans un sous-arbre gauche, et joint_droit après l'insertion dans un sous-arbre droit. Les deux algorithmes sont symétriques et consistent à appliquer une rotation appropriée si nécessaire, nous ne donnons que l'algorithme joint_gauche (Figure 16). La fonction joint permet de s'assurer que tout AVL-arbre vérifie l'invariant (12) concernant l'exactitude du contenu du champ hauteur.

Lemme 2.2. *L'algorithme joint_gauche, avec les préconditions* $\mathsf{hauteur}(g) - \mathsf{hauteur}(d) \in \{-1, 0, 1, 2\}$ *et* $[\![g]\!] < n < [\![d]\!]$, *est correct.*

Démonstration. L'algorithme étant élémentaire (pas de boucle, pas d'appels récursifs) il suffit de prouver que les valeurs retournées vérifient les invariant d'AVL-arbres.

Si le résultat est retourné en ligne 13, par la précondition $\mathsf{hauteur(d)} - \mathsf{hauteur(g)} \leq 1$ et par le test de la ligne 12, $\mathsf{hauteur(g)} - \mathsf{hauteur(d)} \leq 1$, donc $|\mathsf{hauteur(g)} - \mathsf{hauteur(d)}| \leq 1$. L'ordre des nœuds est clairement respecté grâce à la deuxième précondition, c'est l'invariant (11). L'invariant (12) est garanti par l'appel à joint.

Si le résultat est retourné à la ligne 15, nous savons par la conditionnelle de la ligne que $\mathsf{hauteur(g)} - \mathsf{hauteur(d)} = 2$. Par l'invariant (13),

$$\mathsf{hauteur(g.sous\_arbre\_droit)} = \mathsf{hauteur(g)} - 2 + \epsilon = \mathsf{hauteur(d)} + \epsilon \text{ avec } \epsilon \in \{0, 1\}$$

L'appel $\mathsf{joint(g.sous\_arbre\_droit, r, d)}$ produit donc bien un AVL-arbre (puisque aussi $[\![\mathsf{g}]\!] < [\![r]\!] < [\![\mathsf{d}]\!]$) de hauteur $\mathsf{hauteur(g)} - 1 + \epsilon$. Par la condition de la ligne 14, $\mathsf{hauteur(g.sous\_arbre\_gauche)} = \mathsf{hauteur(g)} - 1$, donc l'arbre retourné par la ligne 15 vérifie bien l'invariant d'équilibre (13). L'invariant (11) sur l'ordre des nœuds

```
1   fonction hauteur(avl_arbre arbre) : entier =
2     si arbre = 0 alors retourner −1 sinon retourner arbre.hauteur
3
4   fonction joint(avl_arbre g, t r, avl_arbre d) : avl_arbre =
5     retourner {
6           sous_arbre_gauche = g,
7           sous_arbre_droit = d,
8           contenu = r,
9           hauteur = max{hauteur(g), hauteur(d)}}
10
11  fonction joint_gauche(avl_arbre g, t r, avl_arbre d) : avl_arbre =
12    si hauteur(g) − hauteur(d) ≤ 1 alors
13      retourner joint(g, r, d)
14    sinon si hauteur(g.sous_arbre_gauche) = hauteur(g) − 1 alors   // Cas 1, rotation simple
15      retourner joint(g.sous_arbre_gauche, g.contenu, joint(g.sous_arbre_droit, r, d)))
16    sinon   // Cas 2: rotation double
17      retourner joint(
18        joint(g.sous_arbre_gauche, g.contenu, g.sous_arbre_droit.sous_arbre_gauche),
19        g.sous_arbre_droit.contenu
20        joint(g.sous_arbre_droit.sous_arbre_droit, r, d))
```

FIGURE 16 – Code des constructeurs d'AVL-arbres utilisés dans l'algorithme d'insertion.

est lui aussi validé, car puisque g est un AVl-arbre, $[\![\text{g.sous\_arbre\_gauche}]\!] < [\![\text{g.contenu}]\!] < [\![\text{g.sous\_arbre\_droit}]\!]$ et par la précondition $[\![\text{g.contenu}]\!] < n < [\![\text{g}]\!]$.

Enfin si le résultat est retourné à la ligne 17, par les tests de la ligne 12 et 14, nous savons que

$$\begin{aligned}\text{hauteur(g)} &= \text{hauteur(d)} + 2\\ \text{hauteur(g.sous\_arbre\_gauche)} &= \text{hauteur(g)} - 2 = \text{hauteur(d)}\end{aligned}$$

et par la définition des hauteurs (invariant (12)) appliqué à g,

$$\text{hauteur(g.sous\_arbre\_droit)} = \text{hauteur(g)} - 1 = \text{hauteur(d)} + 1$$

Les deux sous-arbres de g.sous_arbre_droit ont donc pour hauteur hauteur(d) ou hauteur(d) − 1. Nous en déduisons que l'arbre retourné possède des sous-arbres équilibrés et de hauteur hauteur(d) + 1, donc respecte l'invariant (13). Enfin, l'invariant (11) peut se vérifier en l'appliquant sur g, d et g.sous_arbre_droit, ce qui donne :

$$\begin{aligned}[\![\text{g.sous\_arbre\_gauche}]\!] < [\![\text{g.contenu}]\!] < [\![\text{g.sous\_arbre\_droit.sous\_arbre\_gauche}]\!] < [\![\text{g.sous\_arbre\_droit.contenu}]\!]\\ < [\![\text{g.sous\_arbre\_droit.sous\_arbre\_droit}]\!] < [\![\text{r}]\!] < [\![\text{d}]\!]\end{aligned}$$

Nous en déduisons facilement que l'AVL-arbre retourné satisfait l'invariant (11). □

```
1  fonction insère(t elt, avl_arbre arbre) : avl_arbre =
2    si arbre = ⊥ alors retourner joint(⊥, elt, ⊥)
3    sinon si arbre.contenu = elt alors retourner arbre
4    sinon si elt < arbre.contenu alors
5      retourner joint_gauche(insère(elt, arbre.sous_arbre_gauche), arbre.contenu, arbre.sous_arbre_droit)
6    sinon
7      retourner joint_droit(arbre.sous_arbre_gauche, arbre.contenu, insère(elt, arbre.sous_arbre_droit))
```

FIGURE 17 – Insertion dans un AVL-arbre

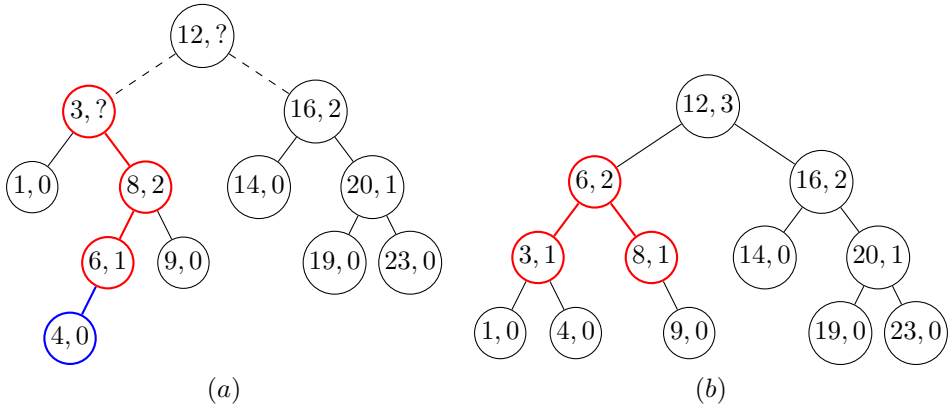


FIGURE 18 – Exemple d'insertion dans un AVL-arbre, l'insertion de 4 provoque une rotation double au nœud 3.

L'algorithme d'insertion est maintenant trivial à écrire. Sa correction est un corollaire du Lemme 2.2 et du fait que l'insertion dans un arbre augmente d'au plus 1 sa hauteur. Cette dernière propriété de l'insertion se prouve par récurrence en analysant la hauteur de l'arbre retourné par les opérations de joint. En effet, par exemple dans le cas d'une insertion à gauche (appel de joint_gauche), si joint_gauche effectue une rotation, la hauteur de l'arbre avant insertion était égale à la hauteur du sous-arbre droit plus 2, et par la preuve du Lemme 2.2 la hauteur de l'arbre retourné par joint_gauche est au plus hauteur(d) + 3.

Enfin la complexité est dominée par le nombre d'appels récursifs de la fonction insertion, qui est bornée par la hauteur de l'arbre. Par le Lemme 2.1, l'insertion dans un arbre à n nœuds est de complexité $O(\log n)$.

Exemple 2.2 (Insertion dans un AVL-arbre). Considérons l'arbre (a) de la Figure 18, dans chaque nœud est noté son contenu et la hauteur du sous-arbre enraciné à ce nœud.

En premier lieu, l'algorithme descend récursivement depuis la racine jusqu'à une feuille, en tenant compte du contenu du nœud traversé pour choisir de descendre vers la droite ou vers la gauche, de même que dans l'algorithme d'appartenance. Il insère alors le nœud 4 à la place de cette feuille.

Les appels récursifs se terminent alors l'un après l'autre, l'algorithme remonte donc dans l'arbre en recalculant à chaque étape la hauteur du nœud, et en vérifiant l'invariant des AVL-arbres (13). Si cet invariant est cassé, l'algorithme, par l'intermédiaire de joint_gauche ou joint_droit exécute une des rotations. Dans cet exemple, au nœud contenant 3, les deux sous-arbres ont pour hauteur 0 et 2, l'invariant est donc violé. Puisque à ce niveau, 4 a été inséré à droite, nous effectuons une rotation droite. Puisque le sous-arbre

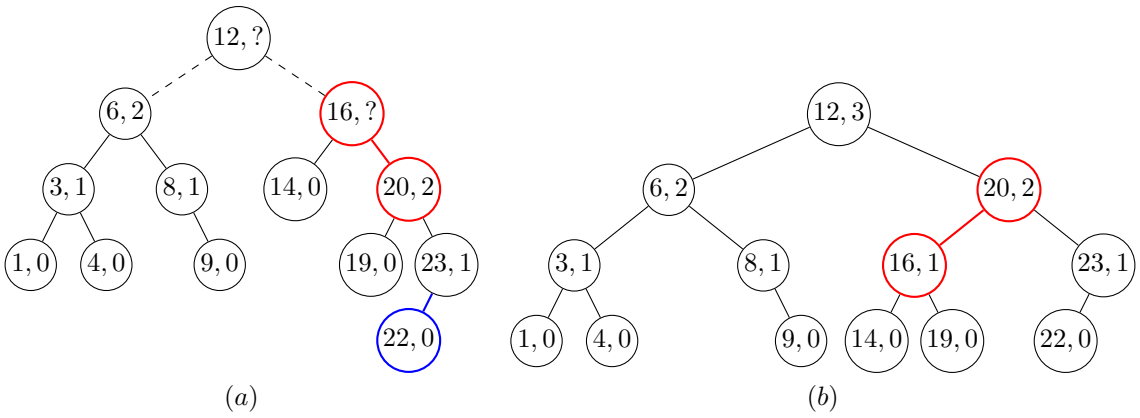


FIGURE 19 – Exemple d'insertion dans un AVL-arbre, l'insertion de 4 provoque une rotation double au nœud 3.

droit du nœud 3 a pour hauteur 2 de plus que son propre sous-arbre droit (enraciné en 9), il s'agit d'une rotation double. Seuls les trois nœuds rouges sont modifiés. Une fois cette rotation effectué, l'algorithme reprend sa remontée vers la racine en mettant les hauteurs à jour.

Continuons l'exemple en insérant le nouvel élément 22, Figure 19. À nouveau, l'algorithme descend dans l'arbre et substitue 22 à une feuille, puis remonte en mettant à jour les hauteurs des nœuds sur le chemin parcouru lors de la descente. Au nœud 16, l'invariant des AVl-arbres (13) est violé. À ce nœud, 22 a été inséré à droite, l'algorithme effectue donc une rotation droite. Puisque la hauteur du sous-arbre droit est de 1 de plus que celle de son propre fils droit, c'est une rotation simple. Une fois la rotation effectué, l'algorithme continue de remonter jusqu'à la racine, sans trouver de nœuds en déséquilibre.

2.1.4 Suppression dans un AVL-arbre

La dernière opération d'un dictionnaire est la suppression d'un élément. La principale difficulté est de combler le vide laisser par cet élément : en effet, le père se retrouve avec un seul fils de moins, mais nous avons deux sous-arbres à rattacher à ce père.

La technique générale consiste donc, plutôt que de supprimer un élément au milieu de l'arbre, simplement de le remplacer par un autre élément, pris ailleurs. Afin de respecter l'invariant des arbres binaires de recherche, il nous faut un élément plus grand que ceux du fils gauche et plus petit que ceux du fils droit. Un bon candidat pour cela est de prendre le plus grand élément du fils gauche.

Il se peut néanmoins que le sous-arbre gauche soit vide, dans ce cas, la suppression du nœud est facile : il suffit de recoller le père du nœud supprimé avec le sous-arbre droit du nœud supprimé.

Enfin, pour s'assurer de respecter l'invariant d'équilibrage des AVL-arbres 13, nous utilisons les opérations joint_droit et joint_gauche lors d'une suppression à gauche ou à droite, respectivement. En effet, une suppression à droite peut baisser d'au plus 1 la hauteur du sous-arbre droit, et une suppression à gauche peut baisser d'au plus 1 la hauteur du sous-arbre gauche (ce qui nécessitera une démonstration).

L'algorithme est décrit en Figure 20. Nous utilisons une fonction auxiliaire pour supprimer le plus grand élément d'un arbre. Par définition, cet élément maximum a un sous-arbre droit vide ce qui rend sa suppression

```
1   fonction supprime_max(avl_arbre arbre) : t × avl_arbre =
2     si arbre.sous_arbre_droit = ⊥ alors
3       retourner (arbre.contenu, arbre.sous_arbre_gauche)
4     sinon
5       soit (max, d) = supprime_max(arbre.sous_arbre_droit)
6       retourner (max, joint_gauche(arbre.sous_arbre_gauche, arbre.contenu, d)
7
8   fonction supprime(t elt, avl_arbre arbre) : avl_arbre =
9     si arbre = ⊥ alors retourner ⊥
10    sinon si elt = arbre.contenu alors
11      si arbre.sous_arbre_gauche = ⊥ alors
12        retourner arbre.sous_arbre_droite
13      sinon
14        soit (m, g) = supprime_max(arbre.sous_arbre_gauche)
15        retourner joint_droite(g, m, arbre.sous_arbre_droit)
16    sinon si elt < arbre.contenu alors
17      retourner joint_droite(supprime(elt, arbre.sous_arbre_gauche), arbre.contenu, arbre.sous_arbre_droit)
18    sinon
19      retourner joint_gauche(arbre.sous_arbre_gauche, arbre.contenu, supprime(elt, arbre.sous_arbre_droit))
```

FIGURE 20 – Suppression d'un élément dans un AVL-arbre

facile. Le test de la ligne 9 permet à l'algorithme de retourner l'arbre initial si l'élément à supprimer n'est pas présent dans l'arbre. Les lignes 16 à 19 correspondent à la recherche de l'élément à supprimer. Les lignes 10 à 15 gèrent la suppression, en appelant la fonction auxiliaire si besoin.

Correction : En dehors des cas de bases, pour lesquels la correction est immédiate, nous devons vérifier les préconditions des fonctions joint_gauche et joint_droit. Pour cela, il suffit de montrer que supprime(elt, arbre) et supprime_max(arbre) retournent un arbre de hauteur au moins $h(\text{arbre}) - 1$. Nous le prouvons par récurrence sur la hauteur de arbre. Supposons que cette propriété soit vrai pour les arbres de hauteurs inférieures à $h(\text{arbre})$. En particulier elle est vrai pour les sous-arbres de arbre. En ligne 3, nous avons clairement, par définition de la hauteur d'un arbre,

$$h(\text{arbre.sous\_arbre\_droit}) = -1 \leq h(\text{arbre.sous\_arbre\_gauche}) = h(\text{arbre}) - 1$$

De même en ligne 12, l'arbre retourné a une hauteur de $h(\text{arbre}) - 1$. En ligne 9, l'arbre retourné vérifie aussi la propriété. Il reste donc les cas où l'arbre retourné est obtenu par joint_gauche ou joint_droit. Dans ces cas, par récurrence, seul l'un des deux sous-arbres passés en argument voit sa hauteur diminuée, d'au plus 1. Les faits suivants sont facilement vérifiable. En absence de rotation, la hauteur de l'arbre retourné ne peut donc diminuer que d'au plus 1. En cas de rotation simple, elle peut être égale ou inférieure de 1 à $h(\text{arbre})$, et en cas de rotation double elle est de $h(\text{arbre}) - 1$.

Enfin, en utilisant l'invariant (11), il est facile de vérifier que les arguments passés à joint_gauche et joint_droit sont correctement ordonnés. Ainsi les préconditions de ces deux fonctions sont satisfaites, donc les arbres construits sont des AVL-arbres valides.

Complexité : En dehors des appels récursifs, toutes les opérations sont élémentaires et prennent donc un temps constant, au plus k. De plus chaque appel récursif se fait sur un arbre de hauteur strictement plus

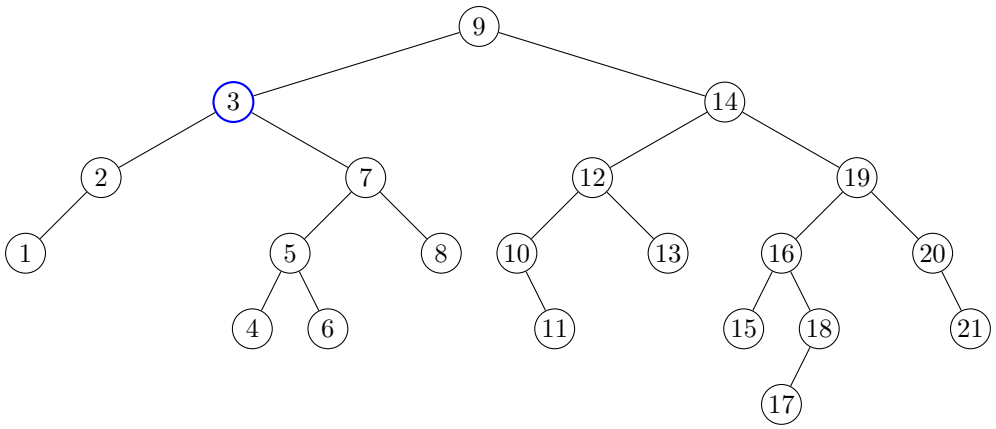


FIGURE 21 – Un AVL-arbre, avant la suppression du nœud 3

petite. La complexité de la suppression dans un arbre T est donc au plus $k \cdot h(T)$. Par le Lemme 2.1, supprime a une complexité asymptotique dans le pire des cas de $O(\log n)$ pour un arbre à n nœuds.

Exemple 2.3 (Suppression dans un AVL-arbre). La première étape consiste à descendre dans l'arbre jusqu'à l'élément à supprimer, ici, le nœud 3 dans l'arbre de la Figure 21. Puis nous cherchons l'élément maximum du sous-arbre gauche de 3, qui est 2, et nous le remplaçons par son propre sous-arbre gauche, par maximalité son sous-arbre droit est vide. Puis nous remplaçons 3 par 2, ce qui donne l'arbre de la Figure 22 (notons que le nœud racine 9 n'est pas encore reconstruit).

La deuxième étape commence, le rééquilibrage de l'arbre obtenu, en remontant depuis le nœud modifié, 2, jusqu'à la racine. Ici, l'invariant des AVL-arbres (13) est rompu au nœud 2, et joint_droit exécute une rotation double droite. Nous obtenons l'arbre de la Figure 23, et nous remontons vers la racine, qui est aussi déséquilibrée.

Cette fois l'algorithme effectue une rotation droite simple, et retourne l'AVL-arbre de la Figure 24, ce qui termine la suppression du nœud 3.

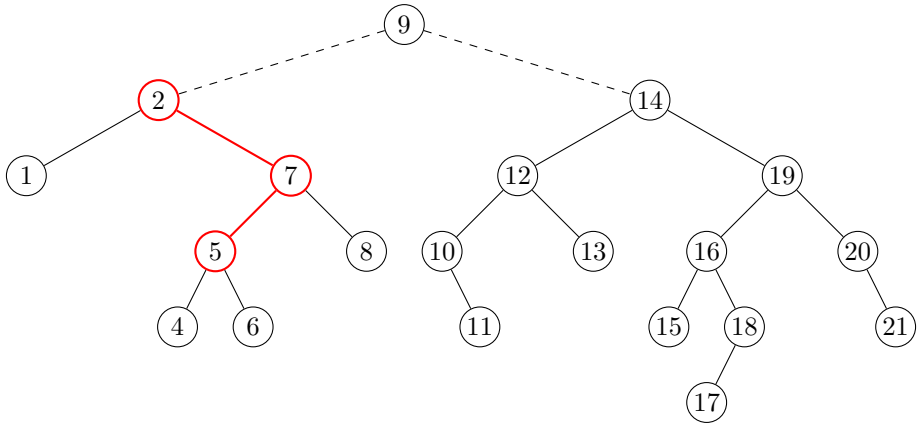


FIGURE 22 – Remplacement du nœud 3 par le maximum de son sous-arbre gauche, une rotation double droite s'impose

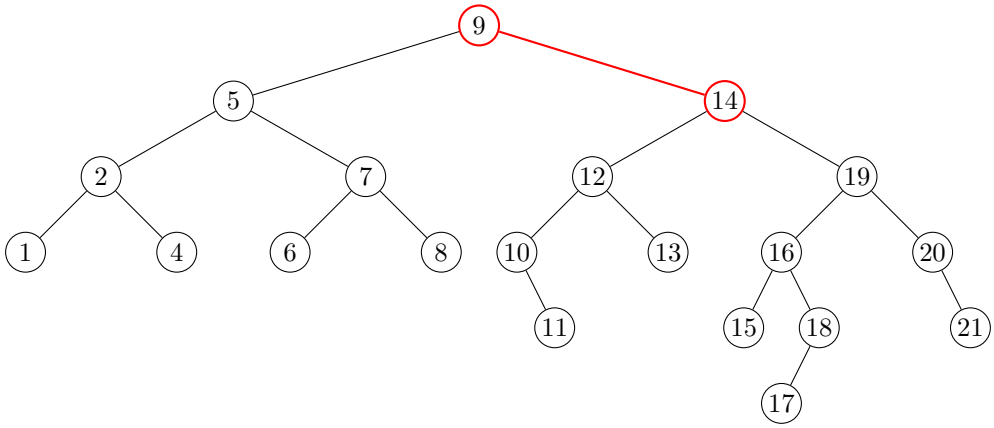


FIGURE 23 – Après rotation du sous-arbre gauche, la racine est déséquilibrée, et nécessite une rotation droite simple.

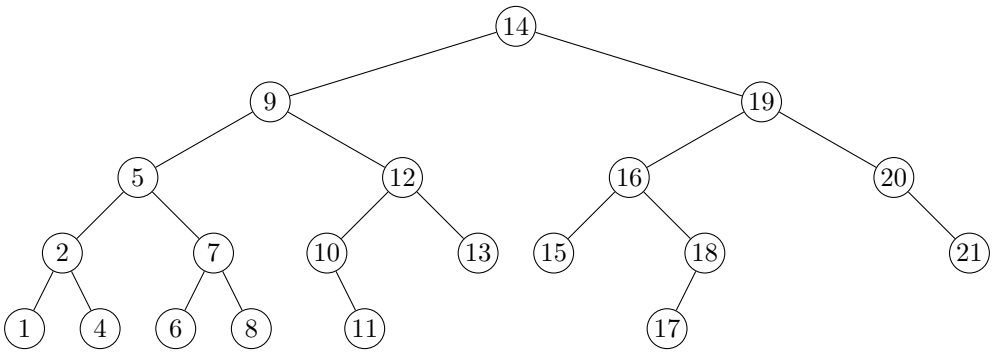


FIGURE 24 – L'arbre final obtenu.

type *tas* **de** *t* **avec fonction** $\leq(t,t)$: *bool*	multi-ensemble d'éléments de type *t*
$\mathsf{tas\_vide}$: *tas*	$[\![\mathsf{tas\_vide}()]\!] = \varnothing$
$\mathsf{est\_vide}(tas)$: *bool*	$[\![\mathsf{est\_vide}(\mathsf{h})]\!]$ = vrai si $[\![\mathsf{h}]\!] = \varnothing$, $[\![\mathsf{est\_vide}(\mathsf{h})]\!]$ = faux sinon
$\mathsf{insère}(t, tas)$: *tas*	$[\![\mathsf{insère}(\mathsf{elt},\mathsf{h})]\!] = \{\mathsf{elt}\} \cup [\![\mathsf{h}]\!]$
$\mathsf{minimum}(tas)$: *t*	$[\![\mathsf{minimum}(\mathsf{h})]\!] = \min([\![\mathsf{h}]\!])$, Précondition $[\![\mathsf{h}]\!] \neq \varnothing$
$\mathsf{extrais\_min}(tas)$: *tas*	$[\![\mathsf{extrais\_min}(\mathsf{h})]\!] = [\![\mathsf{h}]\!] \setminus \{\min([\![\mathsf{h}]\!])\}$ Précondition : $[\![\mathsf{l}]\!] \neq \varnothing$
$\mathsf{union}(tas, tas)$: *tas*	$[\![\mathsf{union}(\mathsf{tas}_1,\mathsf{tas}_2)]\!] = [\![\mathsf{tas}_1]\!] \cup [\![\mathsf{tas}_2]\!]$

FIGURE 25 – La structure de donnée abstraite *file de priorité fusionnable*.

2.2 Files de priorité fusionnables

Nous reprenons la structure de donnée abstraite de file de priorité, et nous l'enrichissons d'une opération supplémentaire, l'union de deux files de priorité.

Définition 2.6. *Une file de priorité fusionnable est une structure de donnée abstraite, détaillée en Figure 25.*

Si nous voulions réutiliser la structure concrète de tas binaire et y ajouter une opération de fusion, la meilleure solution (à un facteur constant près asymptotiquement) consisterait à entièrement créer un nouveau tas binaire, ce qui prendrait un temps $O(n)$ si n est le nombre d'éléments dans l'union. Dans cette section, nous donnons une structure de donnée concrète, les tas gauches, qui permet d'obtenir une opération d'union en temps $O(\log n)$, tout en conservant les mêmes performances que les tas binaires pour les autres opérations. Comme les AVL-arbres, notre implantation utilise les arbres binaires avec des propriétés bien choisies, couplés à une technique de rotation.

Définition 2.7. *Le* rang droit *d'un arbre binaire T, noté $rang(T)$, est défini par $rang(\perp) = -1$, et $rang(T) = 1 + rang(T_d)$ si $T = (n, T_g, T_d)$.*

Un arbre gauche *est un arbre binaire tel que pour tout nœud (n, T_g, T_d), $rang(T_g) \geq rang(T_d)$.*

Le rang correspond donc à la longueur du chemin partant de la racine et descendant toujours vers la droite. La Figure 26 montre un exemple d'arbre gauche.

Lemme 2.3. *Si T est un arbre gauche à n éléments, $rang(T) \leq \log_2(1+n) - 1$.*

Démonstration. Comme dans la preuve du Lemme 2.1 sur la hauteur des AVL-arbres, nous commençons par décrire un arbre de rank r avec un nombre minimum n_r de nœuds. Le sous-arbre droit de sa racine a, par définition du rang, un rang de $r-1$. Par définition des arbres gauches, son sous-arbre gauche aussi doit avoir rang $r-1$. Nous en déduisons :

$$n_r = 2 \cdot n_{r-1} + 1$$

De plus $n_0 = 1$. Le terme général de cette suite est $n_r = 2^{r+1} - 1$. Par croissance du logarithme, $\log_2(1+n_r) = r+1$, d'où le lemme. □

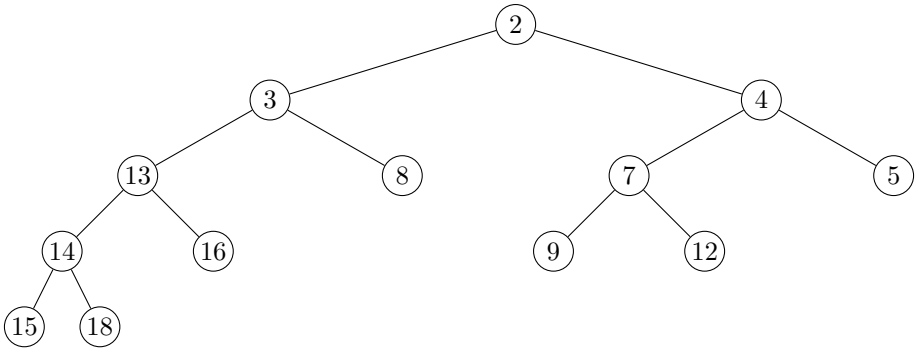


FIGURE 26 – Un tas gauche.

En fait, l'arbre de rang r minimisant le nombre de nœuds est l'arbre binaire complet de hauteur r.

Définition 2.8. *Un* tas gauche *est un arbre gauche tel que pour tout nœud n, si n n'est pas la racine $n \geq parent(n)$.*

Cette propriété est commune avec les tas binaires : les éléments augmentent lorsque nous descendons dans le tas. La différence entre tas binaire et tas gauche consiste donc dans la forme de l'arbre binaire : les tas gauche ont une forme plus libre, et c'est ce qui nous permet de proposer une opération d'union plus efficace que dans les tas binaires.

Pour coder les tas gauches, nous pouvons directement réutiliser le type de donnée des arbres binaires, en le renommant et en ajoutant un champ pour enregistrer le rang d'un nœud.

```
1  type tas_gauche de t =
2    struct{
3      sous_arbre_gauche : tas_gauche;
4      contenu : t;
5      sous_arbre_droit : tas_gauche;
6      rang : entier;
7    } ou bien ⊥
```

La sémantique est définie par :

$$[\![\mathsf{tas}]\!] = \begin{cases} \varnothing & \text{si } \mathsf{tas} = \bot \\ [\![\mathsf{tas.sous\_arbre\_droit}]\!] \cup \{[\![\mathsf{tas.contenu}]\!]\} \cup [\![\mathsf{tas.sous\_arbre\_gauche}]\!] & \text{sinon} \end{cases}$$

Pour simplifier l'écriture des invariants, nous définissons la fonction rang retournant le rang d'un tas gauche :

```
1  fonction rang(tas_gauche tas) : entier =
2    si tas = ⊥ alors retourner −1
3    sinon retourner tas.rang
```

Nous exprimons maintenant les invariants que nous souhaitons sur notre type de donnée. Si tas : *tas_gauche* est différent de $\perp$:

$$\text{tas.rang} = 1 + \text{rang}(\text{tas.sous\_arbre\_droit}) \tag{14}$$

$$\text{rang}(\text{tas.sous\_arbre\_gauche}) \geq \text{rang}(\text{tas.sous\_arbre\_droit}) \tag{15}$$

$$\text{tas.contenu} \leq [\![\text{tas.sous\_arbre\_gauche}]\!] \cup [\![\text{tas.sous\_arbre\_droit}]\!] \tag{16}$$

Tout comme dans les AVL-arbres, il nous sera utile d'avoir une fonction chargée de joindre deux sous-arbres par un nœud, tout en assurant les invariants (14) et (15).

```
1   fonction joint(tas_gauche gauche, t elt, tas_gauche droit) : tas =
2     si rang(gauche) ≥ rang(droit) alors
3       retourner {
4         sous_arbre_gauche = gauche;
5         contenu = elt;
6         sous_arbre_droit = droit;
7         rang = 1 + rang(droit)}
8     sinon retourner {
9         sous_arbre_gauche = droit;
10        contenu = elt;
11        sous_arbre_droit = gauche;
12        rang = 1 + rang(gauche)}
```

Lemme 2.4. *Si droit et gauche sont des tas gauches valides, et que la précondition* $[\![elt]\!] \leq [\![gauche]\!] \cup [\![droit]\!]$ *est satisfaite, alors joint*(*tas_gauche gauche*, *t elt*, *tas_gauche droit*) *est un tas gauche valide de sémantique* $\{[\![elt]\!]\} \cup [\![gauche]\!] \cup [\![droit]\!]$.

Démonstration. Dans les deux cas de retour, l'invariant (14) est clairement vérifié par les calculs du champ rang. L'invariant (15) s'obtient comme conséquence directe de la condition de la ligne 2. Enfin l'invariant (16) est simplement la précondition. Enfin la sémantique est correcte par définition. □

Nous avons maintenant assez d'éléments pour écrire la fonction d'union. Puisque joint ne demande que des tas gauches valides et un élément plus petit, l'idée est de récupérer l'élément minimum des deux tas, l'une des deux racines, ce qui nous laisse l'autre tas et les deux sous-arbres du minimum (donc 3 tas, un de trop). Comme nous savons que le rang est logarithmique, nous procédons par récurrence sur le sous-arbre droit du minimum, que nous fusionnons avec le deuxième tas, réduisant donc le nombre de tas de 3 à 2, ce qui permet d'appeler joint avec l'élément minimum. L'algorithme est donné en Figure 27.

Comme nous pouvons le voir, une fois l'union codé les autres opérations sont triviales. Nous prouvons donc uniquement la correction de union et sa complexité asymptotique.

Correction : Il suffit de démontrer, en vertu du Lemme 2.4, que la précondition de joint est correcte. Au retour de la ligne 5, tas_1.contenu est plus petit que tout élément de tas_1 par l'invariant des tas gauches (16), et que tout élément de tas_2 par le test de la ligne 4 et à nouveau l'invariant (16). Dans ce cas, la précondition de joint est donc satisfaite. Le cas du retour en ligne 7 est symétrique.

Complexité : toutes les opérations sont élémentaires, à l'exception des appels récursifs des lignes 5 et 7. Clairement au plus un des deux peut avoir lieu. De plus, la somme des rangs droits des deux arbres passés

```
1   fonction union(tas_gauche tas1, tas_gauche tas2) : tas_gauche =
2     si tas1 = ⊥ alors retourner tas2
3     sinon si tas2 = ⊥ alors retourner tas1
4     sinon si tas1.contenu ≤ tas2.contenu alors
5       retourner joint(tas1.sous_arbre_gauche, tas1.contenu, union(tas1.sous_arbre_droit, tas2))
6     sinon
7       retourner joint(tas2.sous_arbre_gauche, tas2.contenu, union(tas2.sous_arbre_droit, tas1))
8
9   fonction insère(t elt, tas_gauche tas) : tas =
10    retourner union(joint(⊥, elt, ⊥), tas)
11
12  fonction minimum(tas_gauche tas) : t =
13    retourner tas.contenu
14
15  fonction extrais_min(tas_gauche tas) : tas_gauche =
16    retourner union(tas.sous_arbre_gauche, tas.sous_arbre_droit)
```

FIGURE 27 – Opérations des tas gauches

en argument de union décroit de 1 à chaque appel récursif. Par le Lemme 2.3, la somme de ces rangs est au plus $2 \log n$, donc la complexité de union sur n éléments est $O(\log n)$.

Exemple 2.4 (Union de tas gauches). Nous illustrons l'algorithme d'union avec un exemple présenté en Figure 28. Puisque la plus petite des deux racines est 2, nous faisons un appel récursif à union avec l'arbre de racine 3 et le sous-arbre droit du nœud 2, de racine 14. Lorsque cette union sera calculée, il suffira de terminer avec une opération joint.

Le premier appel récursif, sur les tas illustrés en Figure 29, se déroule de façon similaire. 3 est la plus petite des deux racines, nous faisons donc un deuxième appel récursif sur le sous-arbre droit de 3 et l'arbre de racine 14. Nous arrivons ainsi à la Figure 30, 9 est la plus petite racine, et un troisième appel récursif entre le sous-arbre vide et l'arbre de racine 14 termine immédiatement en retournant l'arbre de racine 14. Le sous-arbre gauche de l'arbre de racine 9 est vide, donc joint construit un nouvel arbre de racine 9, avec en sous-arbre gauche l'arbre de racine 14 (retourné par le 3$^{\text{e}}$ appel récursif), et en sous-arbre gauche l'arbre vide (de rang moindre). Nous obtenons donc l'arbre de gauche sur la Figure 31.

Ensuite, vient le joint de cet arbre avec celui de racine isolé par le premier appel récursif, avec en racine le minimum du premier appel récursif, 3. L'arbre de racine 5 a le rang le plus élevé, il est placé en sous-arbre gauche de 3.

Finalement l'appel initial exécute le joint entre cet arbre et le sous-arbre de racine 6. Les deux ont même rang, l'arbre de racine 6 est placé par défaut en sous-arbre gauche. Le tas final est donné en Figure 32.

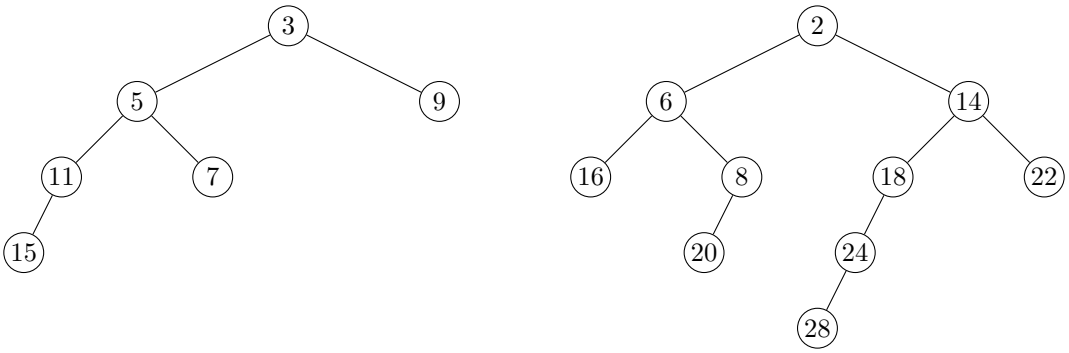


FIGURE 28 – Union de deux tas gauche, état initial

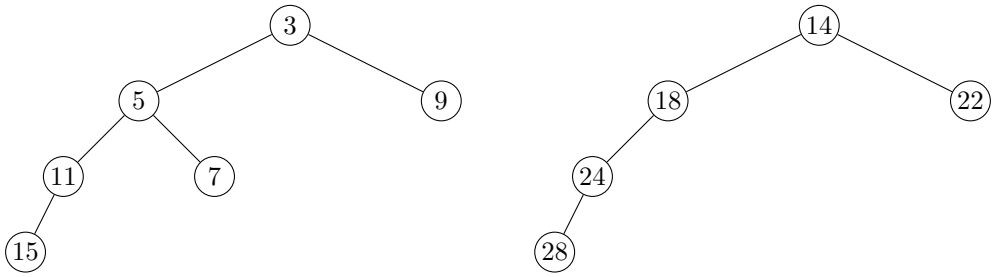


FIGURE 29 – Union de deux tas gauche, premier appel récursif

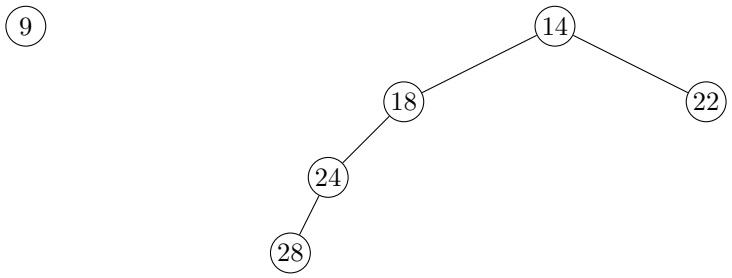


FIGURE 30 – Union de deux tas gauche, deuxième appel récursif

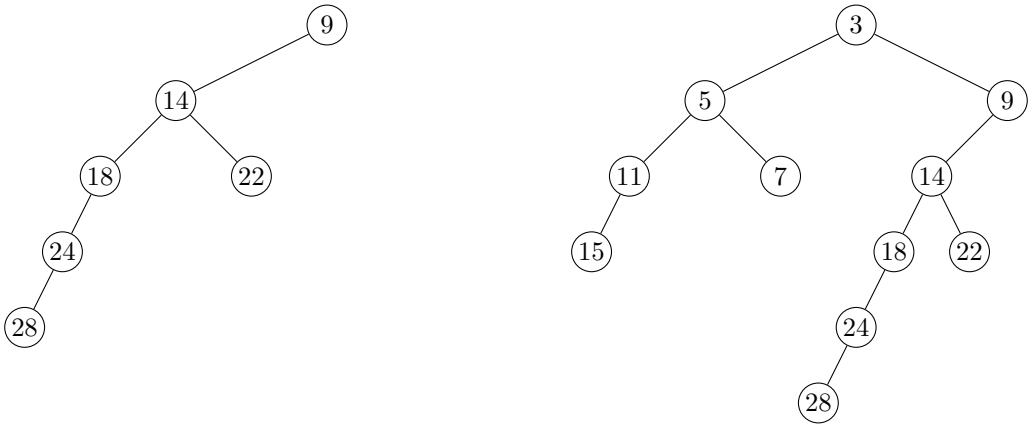


FIGURE 31 – Union de deux tas gauche, retour du deuxième et du premier appel récursif

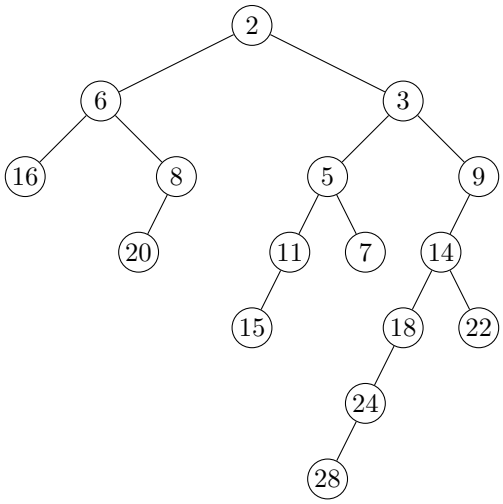


FIGURE 32 – Résultat final de l'union des deux tas.

2.3 B-arbres

Notre modèle habituel de calcul suppose que toute la mémoire est adressable directement au même coût. En pratique, cela est vrai pour une utilisation faible de la capacité de mémoire d'un ordinateur. Le temps d'accès varie énormément entre les caches de différents niveaux du processeur, l'accès à un disque dur, voir un accès à des données stockées dans un réseau. Les B-arbres ont été conçus comme des dictionnaires qui seraient stockés dans une mémoire avec un temps d'accès large, mais un débit élevé, en particulier pour les disques durs. Chaque accès mémoire prend un temps relativement long mais fournit une quantité large (mais constante) d'information. Dans cette section, la complexité sera donc uniquement compté par le nombre d'accès lis_disque et d'écriture écris_disque sur la mémoire secondaire (le disque dur par exemple). Notre modèle de calcul dispose aussi d'une mémoire primaire (par exemple le premier cache), pouvant stocké uniquement un petit nombre de fois la quantité de données retournée par un appel à lis_disque.

Nous allons utiliser un arbre dont chaque nœud a une taille de l'ordre de ce que peuvent traiter lis_disque et écris_disque. Chaque nœud va donc stocker un grand nombre de clés et de sous-arbres, ce n'est pas un arbre binaire.

2.3.1 Définitions

Définition 2.9. *Un* arbre *sur un ensemble de nœud N est ou bien l'arbre vide $\perp$ si $N = \varnothing$, ou bien un uplet $(n, T_1, \ldots, T_k)$ tel que $n \in N$, pour tout $1 \leq i \leq k$, T_i est un arbre sur N_i, et $N_1 \uplus \ldots \uplus N_k = N \setminus \{n\}$. Dans le second cas, k est le* degré *de n, n est la* racine *racine(T) de T, et racine$(T_1), \ldots,$ racine(T_k) sont les* fils *de n. $T_1, \ldots, T_k$ sont les* sous-arbres directs *de T. Les* sous-arbres *de T sont T et les sous-arbres des sous-arbres directs de T.*

La hauteur de T, notée $h(T)$, est définie par $h(\perp) = -1$, $h(n, T_1, \ldots, T_k) = 1 + \max\{h(T_1), \ldots, T_k\}$. La profondeur *d'un nœud n dans l'arbre T est définie comme $h(T) - h(T_n)$, où T_n est le sous-arbre de T de racine n. En particulier, la racine d'un arbre T est l'unique nœud de profondeur 0 dans T.*

Ces définitions sont donc compatibles avec la définition d'arbre binaire, un arbre binaire est un arbre dont tous les nœuds ont degré au plus 2, et les fils d'un nœud sont distingués entre *fils droit* et *fils gauche*.

Pour les B-arbres, chaque nœud va être un sous-ensemble de k mots (ou *clés*) du dictionnaire, et va avoir $k+1$ fils. Afin de respecter la contrainte de quantité d'information lors des accès mémoires, nous bornons k par une constante paire $2B$, de sorte que chaque nœud puisse être lu ou écrit en un seul accès disque. Pour que chaque accès soit efficace, nous voulons aussi que chaque nœud contienne suffisamment d'information, nous imposons donc $k \geq B$. Enfin, puisque nous voulons pouvoir faire des recherches rapidement, nous généralisons l'invariant des arbres binaires de recherche aux B-arbres, et demandons une condition d'équilibrage.

Définition 2.10. *Un B-arbre sur un ensemble ordonné de clés D est un arbre T sur un ensemble de nœuds $N_1, \ldots, N_t$ partition de D, tel que*

(i) chaque nœud autre que la racine contient entre B et $2B$ clés,
(ii) la racine contient au plus $2B$ clés,
(iii) les nœuds vides ont tous la même profondeur,
(iv) Si $(N_i, T_0, \ldots, T_k)$ est un nœud de T, alors $|N_i| = k$, et en notant $n_1 < n_2 < \ldots < n_k$ les éléments de N_i, alors pour tout j, $[\![T_{j-1}]\!] < n_j < [\![T_j]\!]$,

avec la notation $[\![(N, T_0, \ldots, T_k)]\!] = N \cup [\![T_0]\!] \cup \ldots \cup [\![T_k]\!]$ et $[\![\perp]\!] = \varnothing$.

La condition *(iii)* est celle d'équilibrage : tout chemin de la racine vers un nœud vide aura la même longueur. La condition *(iv)* permet de faire la recherche d'un élément efficacement : à chaque nœud, nous pouvons déterminer un unique sous-arbre dans lequel peut se trouver la clé que nous recherchons.

Lemme 2.5. *Si T est un B-arbre sur D avec $n = |D|$, alors $h(T) \leq \log_{B+1}(1+n)$.*

Démonstration. Pour cela, nous considérons les arbres à hauteur fixés contenant un nombre minimum de nœuds. Notons $h = h(T)$. Clairement tous les nœuds sauf la racine ont degré $B+1$, la racine a degré 2. Nous en déduisons aisément que le nombre n de clés est $1 + 2B \cdot \sum_{i=1}^{h-1}(B+1)^i$. En simplifiant la somme géométrique, nous obtenons :

$$n = 1 + 2B \cdot \frac{(B+1)^h - 1}{B} = 2(B+1)^h - 1$$

Puis en passant au logarithme,

$$\log_{B+1}(1+n) \geq h$$

□

2.3.2 Type de donnée et recherche

Pour pouvoir garder chaque nœud sur un espace contigu, ceux-ci sont codés à partir de tableaux. Les clés et les sous-arbres sont stockés dans deux tableaux, de taille $2B$ et $2B+1$. Pour mettre en avant que chaque sous-arbre est ensuite codé dans un espace mémoire propre, nous utilisons explicitement des références sur des nœuds pour coder les arbres (même si les références ne sont pas nécessaire pour spécifier l'implantation). Enfin, nous utilisons un champ booléen est_feuille pour déterminer si un nœud n'a que des sous-arbres vides, et un champ entier longueur pour compter le nombre de clé dans le nœud.

```
1  type noeud de t =
2    struct{
3      cles : tableau de t;
4      fils : tableau de B_arbre de t;
5      degre : entier;
6      est_feuille : bool
7    }
8  type B_arbre de t = ref noeud de t
```

Nous pouvons alors exprimer les invariants nécessaires et la sémantique d'une variable de type *B_arbre*. Notons $l =$!arbre.degre, et h la hauteur de arbre, alors

$$[\![\text{arbre}]\!] = [\![\text{!arbre.fils}[l]]\!] \cup \bigcup_{i=0}^{l-1} (\{[\![\text{!arbre.cles}[i]]\!]\} \cup [\![\text{!arbre.fils}[i]]\!]) \quad \text{si } \neg\text{!arbre.est\_feuille}, \tag{17}$$

$$[\![\text{arbre}]\!] = \bigcup_{i=0}^{l-1} \{[\![\text{!arbre.cles}[i]]\!]\} \quad \text{sinon.} \tag{18}$$

Les invariants sont :

$$\text{!arbre.est\_feuille} \iff h = 0 \tag{19}$$
$$\text{!arbre.degre} \leq 2B \tag{20}$$
$$\forall i \in [0, \text{!arbre.degre}], \text{!arbre.cles}[i] \text{ est défini} \tag{21}$$
$$\forall i \in [0, \text{!arbre.degre} + 1], (\neg \text{!arbre.est\_feuille}) \implies \text{!arbre.fils}[i] \text{ est défini et a hauteur } h - 1 \tag{22}$$
$$\forall i \in [0, \text{!arbre.degre} + 1], (\neg \text{!arbre.est\_feuille}) \implies \text{!}(\text{!arbre.fils}[i]).longueur \geq B \tag{23}$$
$$\forall i \in [0, \text{!arbre.degre} - 1], [\![\text{!arbre.fils}[i]]\!] < [\![\text{!arbre.fils}[i+1]]\!] \tag{24}$$
$$\forall i \in [0, \text{!arbre.degre}], [\![\text{!arbre.fils}[i]]\!] < [\![\text{!arbre.cles}[i]]\!] < [\![\text{!arbre.fils}[i+1]]\!] \tag{25}$$

L'invariant (22) assure que l'arbre est correctement formé et équilibré. Les deux invariants (24) et (25) garantisse que les nœuds sont bien ordonnés dans le B-arbre. Les invariants (20) et (23) disent que chaque nœud (sauf la racine) a bien un degré compris entre B et $2B$. Nous n'imposons pas de borne inférieure sur le degré de la racine, car il n'y a borne inférieur sur un nœud que s'il est fils d'un B-arbre, ce qui n'est pas le cas de la racine.

Nous pouvons alors voir les opérations lis_disque et écris_disque comme des opérations de déréférencement et d'affectation respectivement. Alors écris_disque recopie le premier argument à l'adresse donné par le second :

```
1  fonction lis_disque(B_arbre) : noeud
2  fonction écris_disque(noeud, ref B_arbre) : void
3  fonction alloue_disque(noeud) : B_arbre
```

Nous commençons par la fonction créant un B-arbre vide.

```
1  fonction nouveau_noeud() : noeud =
2    retourner {
3      cles = tableau de taille 2B + 1;
4      fils = tableau de taille 2B + 2;
5      degre = −1;
6
7  fonction empty() : B_arbre =
8    retourner alloue_disque(nouveau_noeud())
```

La recherche d'un élément imite celle des arbres binaires de recherche, la difficulté consiste à trouver le bon fils pour continuer la recherche. Nous faisons une recherche dichotomique dans le tableau correspondant à un nœud, afin de trouver la plus petite clé plus grande ou égale à la clé recherché. En cas d'égalité, nous terminons, sinon, nous continuons la recherche dans le sous-arbre de même indice que cette clé.

```
1   fonction dichotomie(t elt, tableau de t tab, entier l, entier u) : entier =
2     si u ≤ l alors retourner l
3     sinon soit m := ⌊(u + l)/2⌋
4       si tab[m] > elt alors dichotomie(elt, tab, l, m)
5       sinon dichotomie(elt, tab, m, u)
6
7   fonction recherche(t elt, B_arbre arbre) : bool =
8     soit noeud := lis_disque(arbre)
9     soit l := dichotomie(elt, !arbre.cles, 0, !arbre.degre − 1)
10    retourner elt = noeud.cles[l]
11      ∨(¬noeud.est_feuille ∧ elt < noeud.cles[l] ∧ recherche(elt, noeud.fils[l]))
12      ∨(¬noeud.est_feuille ∧ elt > noeud.cles[l] ∧ recherche(elt, noeud.fils[l + 1]))
```

La correction découle des invariants (24) et (25), nous laissons les détails en exercice. La ligne 12 est un peu subtile, elle ne peut servir que si $l = \text{arbre.degre}$, l'élément recherché est plus grand que toutes les clés, puisque sinon l désigne une clé plus grande que elt.

Il est assez clair que le nombre de lecture sur le disque est égal au nombre d'appels récursif, lui-même borné par la hauteur du B-arbre. Par le lemme 2.5, la complexité en nombre d'accès au disque est $O(\log_{B+1} n)$. En terme d'usage du processeur (nombre d'opérations élémentaires), nous devons prendre en compte la recherche dichotomique, qui comme nous le savons utilise $O(\log B)$ opérations élémentaires à chaque nœud, donc La complexité totale est $O(\log_2 B \cdot \log_{B+1} n)$.

2.3.3 Insertion dans un B-arbre

L'insertion se fait toujours dans un nœud le plus profond. La principale difficulté consiste à conserver l'invariant sur le degré d'un nœud. Si l'insertion provoque l'apparition d'un nœud de taille $2B + 1$, il est nécessaire de le couper en 2 nœuds de taille B. Ceci laisse un élément qui permet de départager les deux nouveau nœuds comme fils de leur père commun. Le père commun voit alors son degré augmenté de 1, ce qui peut l'amener a lui aussi devenir de trop haut degré. Le processus peut éventuellement se répéter jusqu'à la racine, dans ce cas nous créons une nouvelle racine de degré 2, et l'arbre gagne 1 en hauteur.

Nous commençons par écrire une procédure pour insérer une clé dans un nœud à un indice donné. Cette procédure prend aussi en argument le sous-arbre devant précéder cette clé, et le sous-arbre devant suivre cette clé. L'ancien sous-arbre se trouvant à cet indice est considéré comme perdu. Notons bien que insère_cle n'assure pas que l'invariant (20) (sur le nombre maximum de clés) est respecté, nous réglerons ce problème plus tard.

```
1   fonction insère_cle(noeud n, t elt, entier indice, B_arbre gauche, B_arbre droit) :  void =
2     soit j := ref n.degre
3     tant que !j > indice faire
4       n.cles[!j] ← n.cles[!j − 1]
5       si ¬n.est_feuille alors n.fils[!j + 1] ← n.fils[!j]
6       j ←  j − 1
7     n.cles[indice] ← elt
8     si ¬n.est_feuille alors n.fils[indice] ← gauche
9     si ¬n.est_feuille alors n.fils[indice + 1] ← droit
10    n.degre ← n.degre + 1
```

Notons que si le degré est initialement de $2B$, l'algorithme alloue une clé à l'indice $2B$, ce qui nécessite un tableau de taille $2B+1$. C'est pourquoi nous avons défini un nœud avec un tableau de clés de taille $2B+1$ et un tableau de fils de taille $2B+2$, alors qu'un de moins aurait du suffire.

Si le degré d'un nœud est égal à $2B+1$, nous devons le couper en deux nœuds, c'est le but de la prochaine fonction. trancher prend un nœud avec $2B+1$ clés, et le partitionne en deux nœuds de B clés, plus une clé intermédiaire. Si $(\mathsf{g}, \mathsf{cle}, \mathsf{d})$ est le résultat de tranche(n), alors $[\![\mathsf{g}]\!] < [\![\mathsf{cle}]\!] < [\![\mathsf{d}]\!]$, et $[\![\mathsf{n}]\!] = [\![\mathsf{g}]\!] \cup \{[\![\mathsf{cle}]\!]\} \cup [\![\mathsf{d}]\!]$. Ces nouveaux nœuds peuvent donc être inséré à la place de l'ancien nœud n.

```
1   type coupe = struct{
2     arbre_gauche : B_arbre;
3     cle : t;
4     arbre_droit : B_arbre} ou bien ⊥
5
6   fonction tranche(noeud n) :  coupe =
7     soit gauche = nouveau_noeud()
8     soit droit = nouveau_noeud()
9     pour tout j entre 0 et B − 1 faire
10      gauche.cles[i] ← noeud.cles[i]
11      droit.cles[i] ← noeud.cles[i + B + 1]
12      gauche.fils[i] ← noeud.fils[i]
13      droit.cles[i] ← noeud.fils[i + B + 1]
14    gauche.fils[B] ← noeud.fils[B]
15    droit.fils[B] ← noeud.fils[2B + 1]
16    gauche.degre ← B; droit.degre ← B
17    gauche.est_feuille ← n.est_feuille
18    droit.est_feuille ← n.est_feuille
19    retourner {arbre_gauche = alloue_disque(gauche),
20               cle = noeud.cles[B],
21               arbre_droit = alloue_disque(droit)}
```

L'insertion peut maintenant être décrite simplement. Nous insérons toute nouvelle clé dans la feuille approprié. Comme cette opération peut provoquer la formation d'un nœud avec $2B+1$ clés, si c'est le cas nous tranchons ce nœud, et dans son père nous le remplaçons par les deux nouveaux nœuds, séparés par la clé intermédiaire que retourne aussi tranche. Pour cela, la fonction auxiliaire insère_interne retourne le triplet obtenu par l'appel à tranche (ligne 12). Après un appel récursif, si l'appel s'est terminé en tranchant un nœud, nous réinsérons le résultat de tranche (ligne 10) : si un fils est devenu trop gros, il s'est fait trancher en deux et nous le remplaçons par les deux nouveaux nœuds. Le processus se répète sur le chemin de récursion, jusqu'à la racine.

```
1   fonction insère_interne(t elt, B_arbre arbre) : coupe
2     soit n := lis_disque(arbre)
3     soit l := dichotomie(elt, n.cles, 0, n.degre)
4     soit i := si elt < n.cles[l] alors l sinon l + 1
5     si n.est_feuille alors
6       insère_cle(elt, i, ⊥, ⊥)
7     sinon
8       soit resultat := insertion_interne(elt, n.fils[i])
9       si resultat ≠ ⊥ alors
10        insère_cle(resultat.cle, i, resultat.gauche, resultat.droite)
11    si n.degre = 2B + 1 alors
12      retourner tranche(n)
13    sinon
14      écris_disque(n, arbre)
15      retourner ⊥
16
17  fonction insère(t elt, B_arbre arbre) : void =
18    soit resultat := insère_interne(elt, arbre)
19    si resultat ≠ ⊥ alors
20      soit n = nouveau_noeud()
21      n.cles[0] ← resultat.cle
22      n.fils[0] ← resultat.arbre_gauche
23      n.fils[1] ← resultat.arbre_droit
24      n.degre ← 1; n.est_feuille ← faux
25      écris_disque(n, arbre)
```

Puisque qu'insertion_interne renvoie parfois un triplet, le code se termine par la fonction insère avec le type voulu, qui utilise insertion_interne. Si la racine est de taille $2B + 1$, elle aussi est coupée en deux.

Exemple 2.5. Nous illustrons cet algorithme par l'insertion de 238 et ensuite de 569 dans le B-arbre partiellement dessiné en Figure 33. Nous fixons la valeur de B à 3, donc chaque nœud, sauf la racine, a entre 3 et 6 clés.

Dans un premier temps, 238 est inséré dans une feuille, en violant éventuellement l'invariant de B-arbres sur la taille des nœuds. C'est le cas ici, puisque la feuille contient ensuite 7 clés de 217 à 240. Cette feuille est donc coupée en deux nœuds séparés par l'élément médian 233, par la fonction trancher. 233 est alors remonté au niveau supérieur est inséré, avec les deux nœuds à sa gauche et à sa droite. L'ajout de la clé 233 à ce nœud ne provoque pas de débordement, le nombre de nœuds est 4, inférieur ou égal à la limite maximum de 6. L'insertion est donc terminée. L'arbre obtenu est donné en Figure 34.

Nous insérons maintenant 569. Dans un premier temps, nous l'insérons dans la feuille contenant les éléments 567 à 574. Comme à la suite de cet ajout elle comporte 7 clés, soit une de trop, elle est tranchée, et l'élément médian 570 remonte dans l'arbre, et est ajouté parmi les clés du père. Nous obtenons l'arbre représenté en Figure 35.

Ceci provoque en réaction une violation de l'invariant de taille des nœuds au nœud contenant les clés 485 à 600. À nouveau ce nœud est tranché en deux nœuds de taille 3 et l'élément médian 565 remonte d'un niveau. C'est la Figure 36.

La racine à son tour dépasse la limite de nombre de clé. Elle est aussi coupé en deux. Cette fois, l'élément médian 343 ne peut être remonté vers un nœud du niveau supérieur, nous créons donc un nouveau nœud ne

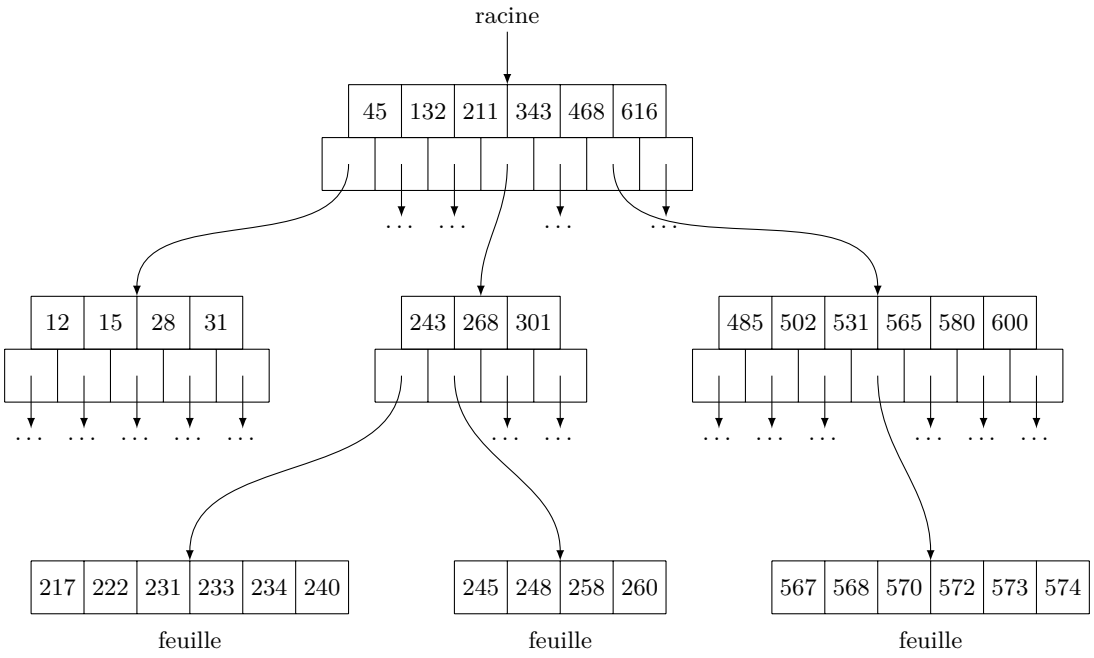


FIGURE 33 – Le B-arbre initial (représentation partielle). Nous avons pris $B = 3$.

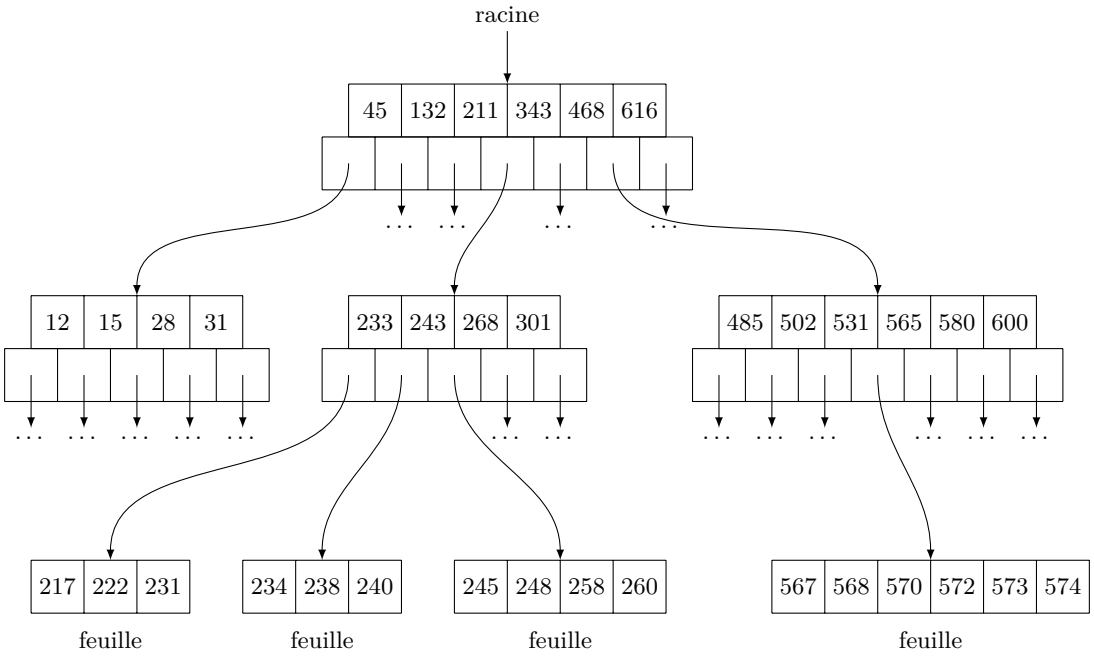


FIGURE 34 – Après l'insertion de 238, la feuille contenait clés et a donc été tranchée en deux. Ceci provoque la remontée de l'élément médian 233.

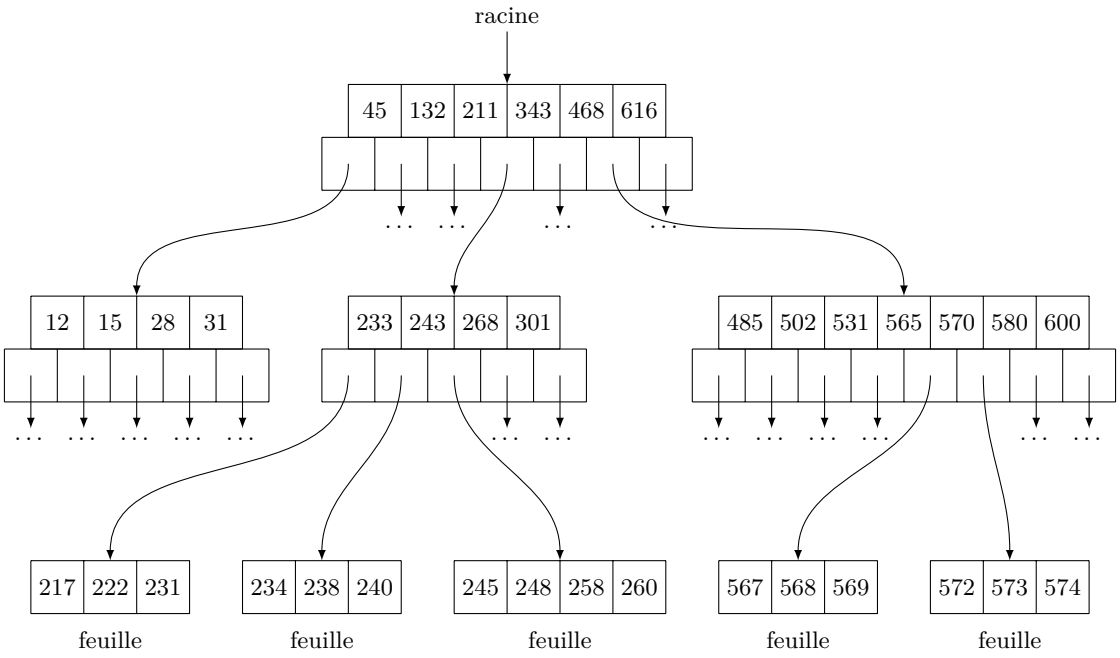


FIGURE 35 – L'insertion de 569 provoque la séparation d'une feuille en deux, et la remontée de 570 vers son père.

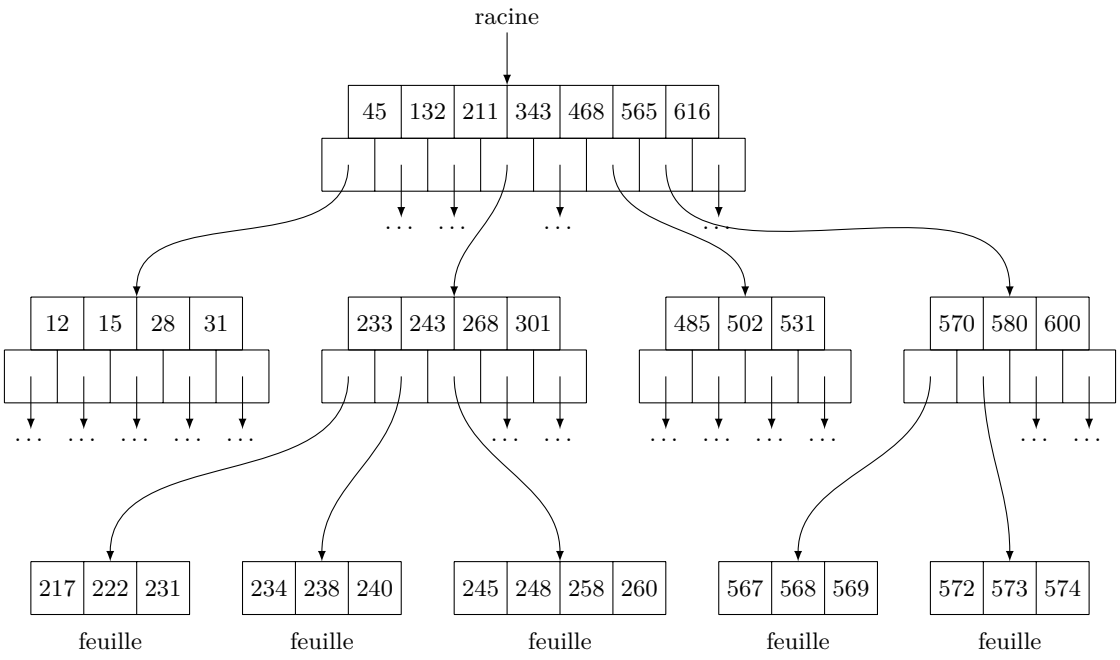


FIGURE 36 – 565 remonte d'un niveau, et est donc inséré dans le nœud racine.

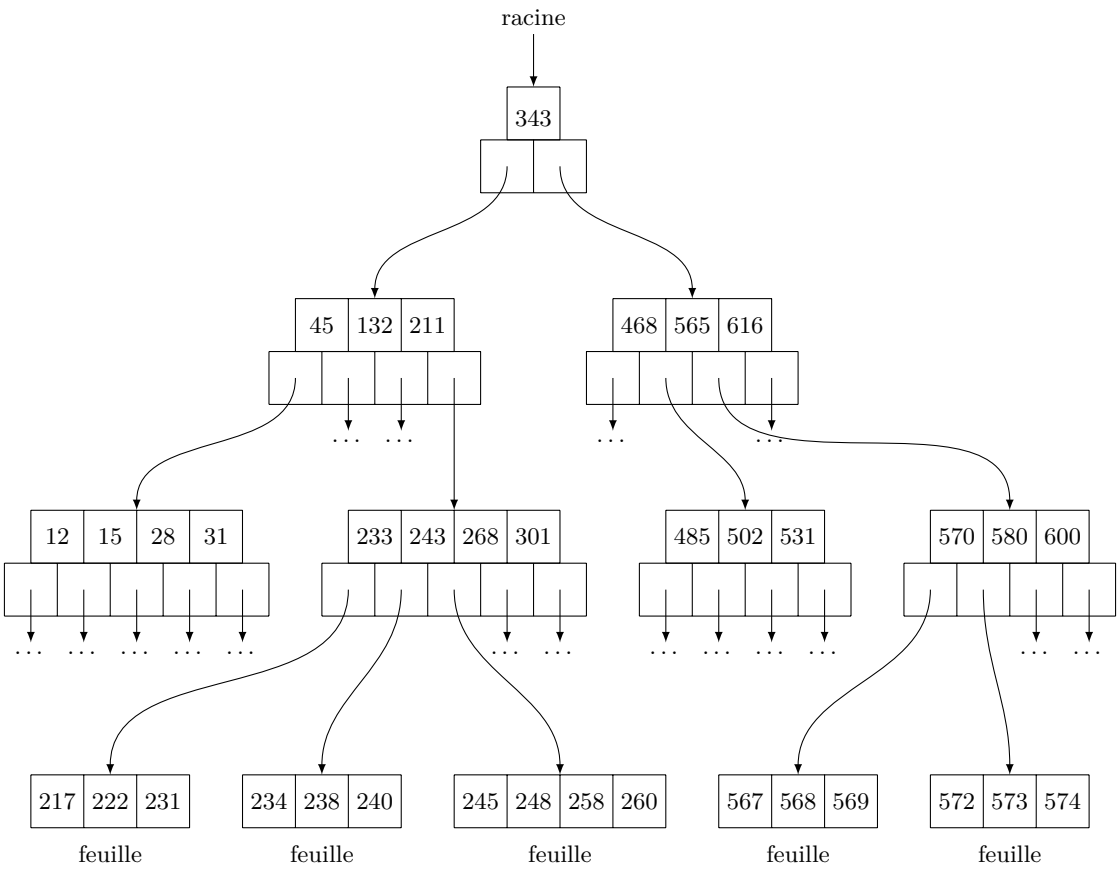


FIGURE 37 – La racine, trop grosse, est coupée en deux, puis l'algorithme termine, nous obtenons cet arbre.

contenant que cette clé. La hauteur de l'arbre augmente de 1. Puis l'algorithme termine, tous les nœuds ont une taille valide, Figure 37.

Complexité : La fonction trancher utilise une boucle **pour tout** exécutant exactement B itérations, chacune prenant un temps constant. Toutes les autres opérations sont des affectations et ont donc globalement aussi un temps constant, donc trancher a une complexité asymptotique $O(B)$. de plus trancher fait deux allocations sur le disque.

Sans compter les appels récursifs, insère_interne fait

— des opérations élémentaires,
— un appel à dichotomie prenant $O(\log_2 B)$,
— un appel à insertion_cle, dont une analyse rapide établie la complexité à $O(B)$ dans le pire des cas,
— au plus un appel à trancher,
— une lecture et un écriture sur le disque.

En plus, insertion_interne peut faire un appel récursif au plus, diminuant la hauteur du B-arbre de 1. Le nombre

d'appel récursif est donc égal à la hauteur du B-arbre qui est $O(\log_{B+1} n)$. En multipliant la complexité pour un seul appel par le nombre d'appel, nous obtenons une complexité totale pour insère de $O(B \cdot \log_{B+1} n)$ avec $O(\log_{B+1} n)$ opérations sur le disque.

2.3.4 Un mot sur la suppression

Supprimer un élément dans un B-arbre est une opération très fastidieuse, qu'aucun livre ne semble avoir le courage de traiter en détail, nous suivrons donc la tradition en ne faisant qu'esquisser les principes.

Comme dans la plupart des implantations d'arbres de recherche, nous commençons par nous ramener à la suppression du minimum ou bien du maximum. En fait plus exactement à la suppression d'une clé d'une feuille. Ainsi, pour supprimer une clé quelconque, nous supprimons au besoin (c'est-à-dire si cette clé n'est pas dans une feuille) la plus petite clé plus grande que la clé à supprimer, qui est nécessairement une feuille. Puis nous remplaçons la clé à supprimer par la clé effectivement supprimée. Ceci permet de ne modifier la structure de l'arbre qu'au niveau d'une feuille.

Lorsque nous supprimons la clé d'une feuille, le seul problème intervient si la feuille ne possédait que B clés. Dans ce cas, il nous faut trouver une autre clé pour ne pas violer la taille minimum d'un nœud. Nous remontons donc dans le nœud père, et cherchons si le fils précédent ou le fils suivant le nœud de taille $B-1$ contient au moins $B+1$ éléments. Si c'est le cas, nous pouvons déplacer une de ces clés parmi les $B+1$ vers le nœud de taille $B-1$. Ceci implique d'utiliser une clé du nœud père pour garder l'invariant sur l'ordre des clés. Ainsi, si nous enlevons 238 de l'arbre de la Figure 37, nous pouvons le remplacer par 243, que nous remplaçons lui-même par 245. C'est (simplement ?) une rotation ! et nous obtenons l'arbre en Figure 38.

Il peut malheureusement arriver que les deux nœuds frères adjacents du nœud de taille $B-1$ soient de taille B. Ce serait le cas si nous supprimons 240 de l'arbre obtenu, Figure 38. Dans ce cas de figure, nous fusionnons le nœud de taille $B-1$ avec un nœud de taille B, en utilisant en plus une clé du père, pour obtenir un nœud de taille $2B$. C'est une opération inverse de trancher. Pour la suppression de 240, nous fusionnons avec le nœud contenant 248,258,260, en utilisant en plus la clé intermédiaire 245 du père. Cela donne l'arbre en Figure 39

Cette opération diminue le nombre des clé dans le nœud père, qui peut à son tour se retrouver avec seulement $B-1$ clés. Dans ce cas, il faut répéter la stratégie précédente, autant de fois que nécessaire ou bien jusqu'à atteindre la racine. Enlever un nœud de la racine est valide tant qu'il reste au moins un nœud ensuite. Pour enlever le dernier nœud de la racine, il suffit de fusionner les deux fils et supprimer la racine, et la hauteur de l'arbre diminue de 1.

Tout cela peut s'implanter avec un peu d'effort de sorte à obtenir une complexité asymptotique de $O(B \log_{B+1} n)$.

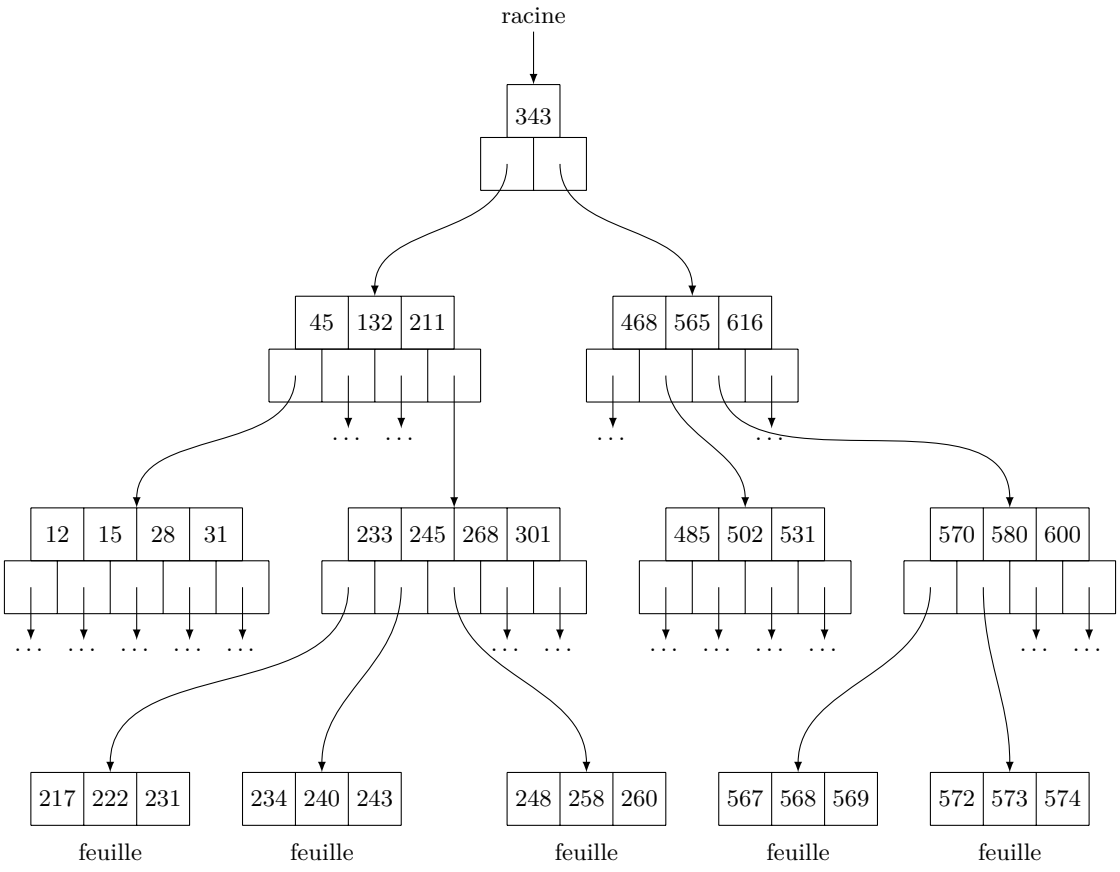


FIGURE 38 – Après la suppression de 238.

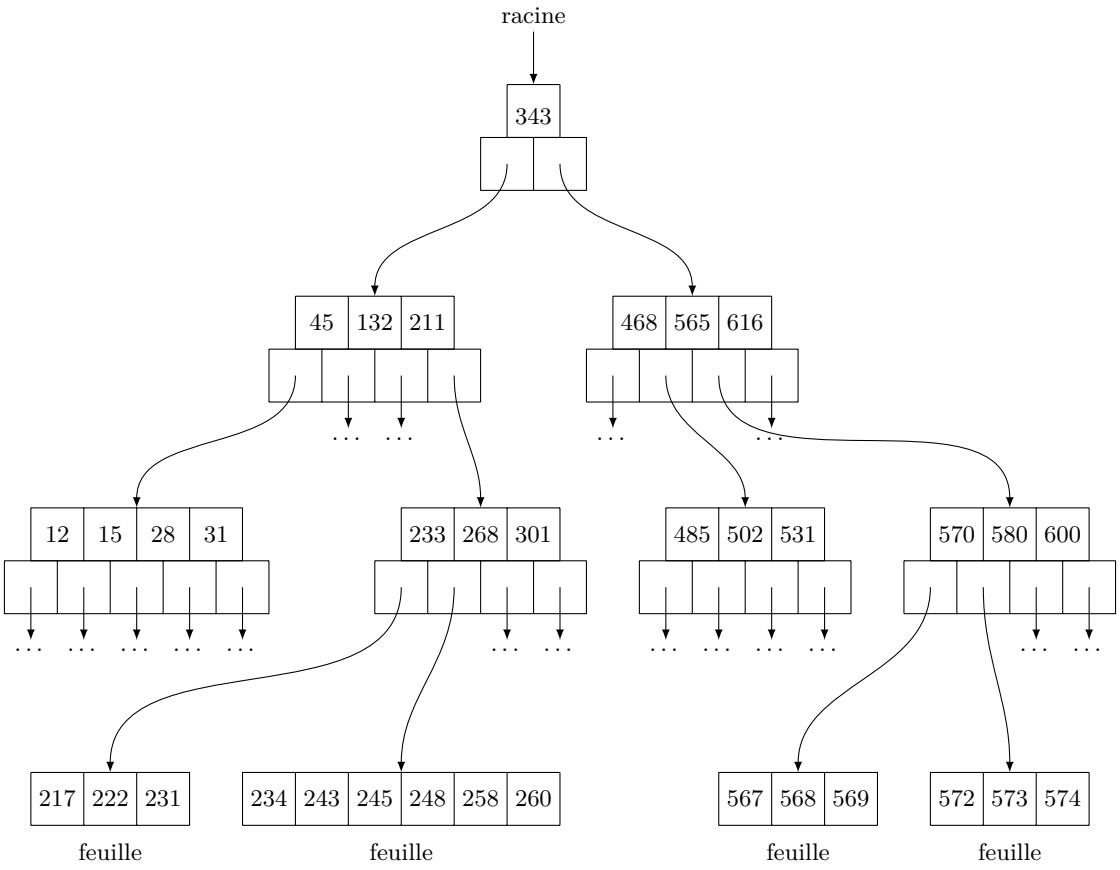


FIGURE 39 – Après la suppression de 240.

2.4 Récapitulatif

Nous avons introduit deux structures de données concrêtes implémentant des dictionnaires, dont l'une adaptée à un usage sur une mémoire distante.

Les AVL-arbres : pour n éléments,
- — recherche en $O(\log n)$ dans le pire de cas,
- — insertion en $O(\log n)$ dans le pire de cas,
- — suppression en $O(\log n)$ dans le pire de cas.

Les B-arbres : pour n éléments,
- — recherche en $O(\log_2 B \cdot \log_{B+1} n)$ dans le pire de cas,
- — insertion en $O(B \log_{B+1} n)$ dans le pire des cas,
- — suppression en $O(B \log_{B+1} n)$ dans le pire des cas.

avec dans chaque cas au plus $O(\log_{B+1} n)$ accès à la mémoire distante.

Nous avons aussi définie une structure de donnée concrête pour les files de priorité fusionnables.

Les tas gauches : pour n éléments,
- — minimum en $O(1)$ (temps constant) dans le pire de cas,
- — insertion en $O(\log n)$ dans le pire de cas,
- — extrait_min en $O(\log n)$ dans le pire de cas,
- — union en $O(\log n)$ dans le pire de cas.

3 Tables à adressage dispersé

Nous avons déjà vu plusieurs structures de donnée concrêtes pour implémenter des dictionnaires. La complexité des opérations que nous obtenions était de $O(\log n)$ pour n clés dans le pire des cas (pour insertion, appartient ou suppression). Pouvons-nous faire mieux ? Si nous supposons que la seule opération autorisée sur les clés est la comparaison de deux clés, la réponse est non, les arbres binaires de recherches sont optimaux. Heureusement, nous savons souvent faire plus de choses sur les clés, et ce chapitre va nous permettre avec des hypothèses plus fortes d'avoir des complexités de $O(1)$ pour chaque opération.

Imaginons un cas simple : nous voulons faire un dictionnaire dont les clés sont les entiers compris entre 1 et N. Pour cela, il suffit de créer un tableau d'indice $[1, N]$ contenant des booléens, indiquant si l'indice est contenu dans le dictionnaire. Clairement les trois opérations peuvent se coder avec une complexité de $O(1)$. Par contre, prenons un cas extrême, supposons que $N = 2^{32}$ (nous voulons encoder un ensemble d'entiers non-signés), mais que le nombre de clés n est petit, disons quelques milliers de clés. Alors cette solution demande plusieurs gigaoctets de mémoire pour coder un tout petit ensemble d'éléments, qui rentrerait dans un AVL-arbre de quelques kilo-octets.

La solution est d'utiliser une fonction $h : [1, N] \to [1, n]$, et de stocker chaque élément e de l'ensemble dans la case du tableau d'indice $h(e)$. En fait cela n'a maintenant plus aucune importance que les clés soient des entiers, tout ce dont nous avons besoin est d'une fonction $h : \{\text{clés}\} \mapsto [1, n]$, que nous appelerons *fonction de dispersion* (ou l'anglicisme *fonction de hachage*).

3.1 Chaînage

À partir de maintenant, nous supposons donc que nous disposons d'un ensemble de toutes les clés possibles $\mathcal{K}$, et pour tout entier n, d'une fonction de dispersion $h_n : \mathcal{K} \mapsto [1, n]$. Nous voulons implémenter un dictionnaire stockant k clés dans un tableau de taille $n = O(k)$ en utilisant h_n pour connaître l'indice d'une clé.

Définition 3.1. *Une* fonction de dispersion *d'ordre n pour un ensemble $\mathcal{K}$ est une fonction $h : \mathcal{K} \mapsto [1, n]$. Étant donné une fonction de dispersion h, nous appelons* indice de dispersion *d'une clé c la valeur $h(c)$.*

Typiquement, les fonctions de dispersion sont choisis pour être uniforme : pour tout $i \in [1, n]$, la probabilité qu'une clé tirée uniformément aléatoirement parmi $\mathcal{K}$ ait un indice de dispersion i est $\frac{1}{n}$:

$$\Pr[h(c) = i] = \frac{1}{n}$$

Nous supposerons que c'est le cas dans nos analyses.

Dans l'idéal, chaque clé aurait pour image par h_n un indice différent. Malheureusement nous ne connaissons pas l'ensemble de k clés à encoder *a priori*, nous devons donc choisir la fonction h_n avant de savoir quelles sont les clés. Puisque le nombre de clés possibles $|\mathcal{K}|$ est généralement plus grand que n, h_n n'est pas une fonction injective, et deux clés peuvent avoir le même indice.

Exemple 3.1. Nous souhaitons coder un annuaire inversé de nos amis :
- Gilles, 04.91.26.00.02
- Michel, 04.91.26.00.03
- Julien, 04.91.26.00.63

Ainsi étant donné un numéro, nous pourrons dire à qui il appartient. Ici $\mathcal{K} = 10^{10}$, $k = 3$, nous prenons $n = 5$ et pour fonction de dispersion $p \mapsto p \mod 5$.

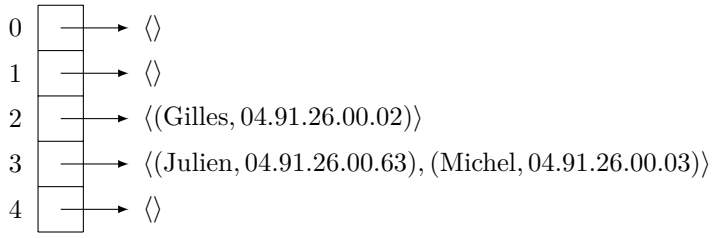
Nous allons donc stocker (Gilles,04.91.26.00.02) en case 2, et (Michel, 04.91.26.00.03) en case 3. Julien et son numéro devraient être stockés en case 3 aussi, mais il n'y a pas la place.

Définition 3.2. *Deux clés c_1, c_2 sont en* collision *pour une fonction de dispersion h si $h(c_1) = h(c_2)$.*

Puisque les collisions sont inévitables, il faut utiliser des techniques supplémentaires pour les gérer. Nous en verrons deux : le chaînage et la méthode du coucou. Nous commençons par le chaînage.

Puisque deux clés ou plus peuvent avoir le même indice, nous pouvons simplement les mettre dans la même case de la table de dispersion. Pour cela, chaque case doit non pas contenir une clé, mais une liste de clés éventuellement vide. Une fois calculé l'indice de dispersion de la clé considérée, nous sommes donc ramener à une opération sur une liste (insertion, appartient ou suppression).

Exemple 3.2. Avec le chaînage, nous obtenons la table :



Lemme 3.1. *Les opérations insertion, appartient et suppression dans une table de dispersion chainée de taille n avec k clés ont une complexité $O(1 + \frac{k}{n})$ en espérance.*

Démonstration. Il s'agit d'une analyse en espérance : nous supposons donc que nous voulons insérer, supprimer ou tester l'appartenance d'une clé c choisie aléatoirement dans $\mathcal{K}$.

Puisque la table T contient k clés et n cases, chaque case contient en moyenne k/n éléments. Donc l'espérance de la longueur de la liste indicée par $h_n(c)$ est elle aussi k/n :

$$\begin{aligned} E[\text{longueur}(T[h_n(c)])] &= \sum_{i=1}^{n} \Pr[h(c) = i] \cdot \text{longueur}(T[i]) \\ &= \sum_{i=1}^{n} \frac{1}{n} \cdot \text{longueur}(T[i]) \\ &= \frac{1}{n} \cdot \left(\sum_{i=1}^{n} \text{longueur}(T[i]) \right) \\ &= \frac{k}{n} \end{aligned}$$

Le lemme est alors prouvé car toutes ces opérations sur les listes sont de complexité linéaire. □

Bien sûr dans le pire des cas toutes les clés auraient le même indice, mais cet événement est terriblement improbable (pour peu que la fonction de dispersion ait été judicieusement choisie). Nous pouvons par contre nous intéresser à la longueur maximum d'une liste dans une table de dispersion. En effet, même en supposant

que la fonction de dispersion est optimale, la taille maximale d'une liste n'est pas une constante, elle est en espérance $O\left(\frac{\log n}{\log \log n}\right)$ lorsque $k = O(n)$. Enfin, nous pouvons borner la probabilité qu'une case contiennent exactement p éléments :

Lemme 3.2.

$$\Pr[\textit{exactement } p \textit{ clés ont indice de dispersion } 1] \leq \frac{\alpha^p}{p!} e^{\frac{p}{n}-\alpha}$$

avec $\alpha = \frac{k}{n}$.

Démonstration. La probabilité que p clés aient le même indice de dispersion est donné par $\binom{k}{p} n^{-p}$: le premier terme correspond au choix de p clés parmi les k disponibles, le second terme correspond à la probabilité que chacune de ces k clés est le même indice de dispersion. Ensuite la probabilité que les autres clés n'aient pas cet indice de dispersion est $\left(1 - \frac{1}{n}\right)^{k-p}$ (chaque indice étant indépendant des autres).

Nous obtenons, en utilisant que $(1 - \frac{1}{n})^n \leq \frac{1}{e}$ pour $n > 0$:

$$\begin{aligned}
\Pr[\text{exactement } p \text{ clés ont indice de dispersion } 1] &= \binom{k}{p} \cdot n^{-p} \cdot \left(1 - \frac{1}{n}\right)^{k-p} \\
&\leq \frac{k^p}{p!} \cdot \left(\frac{1}{n}\right)^p \cdot \left(\left(1 - \frac{1}{n}\right)^n\right)^{\frac{k-p}{n}} \\
&\leq \frac{\alpha^p}{p!} \cdot e^{\frac{p}{n}-\alpha}
\end{aligned}$$

□

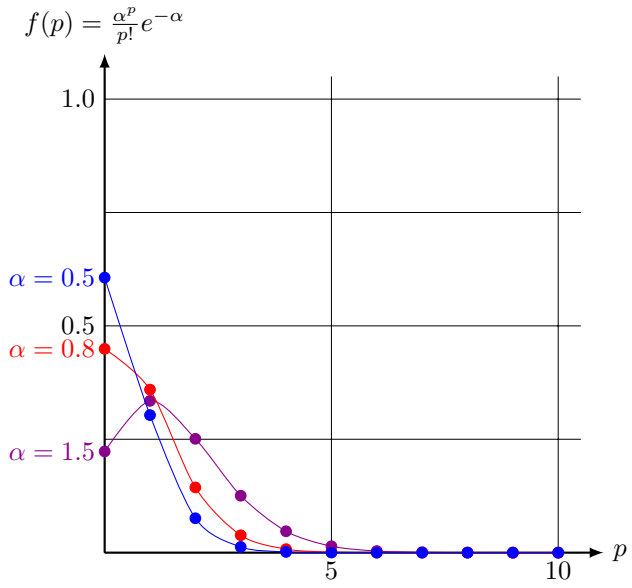
En pratique, pour des valeurs de p très faible devant n, les résultats montrent un très grande proximité entre la borne $\frac{\alpha^p}{p!} e^{-\alpha}$ et les valeurs mesurées. Nous donnons la représentation graphique de cette fonction pour différentes valeurs de α et $p \leq 10$, en Figure 40.

Enfin pour stocker k clés, il est courant de prendre $n = k/\alpha$ avec α de l'ordre de 1, mais il est possible de jouer sur la valeur de α pour avoir des temps d'accès brefs et une grande occupation mémoire (α petit), ou au contraire une occupation mémoire faible pour des temps d'accès un peu plus long (α grand). Par contre prendre α trop grand est contre-productif : la taille des listes finit par dominer la complexité en espace, et cette taille est toujours $O(k)$, donc est indépendante de α.

3.2 Fonctions de dispersion

Nous avons vu que pour obtenir une table de dispersion de bonne qualité, il est nécessaire d'éviter les collisions, et donc d'essayer d'avoir une fonction de dispersion uniforme. Un autre souci est le temps de calcul de cette fonction : comme chaque opération du dictionnaire requiert le calcul d'un indice de dispersion, et que le reste de l'opération se fait en temps constant, le temps d'exécution en pratique est rapidement dominé par le temps du calcul de l'indice si celui-ci n'est pas assez bref.

Plusieurs techniques existent pour construire des fonctions de dispersion, mais en théorie la fonction doit être uniforme par rapport à la distribution de probabilité des clés (qui quand à elle n'est pas nécessairement uniforme). Il n'y a donc pas de fonction de dispersion magique qui marcherait à tous les coups. Nous donnons donc seulement quelques familles de fonctions de dispersion usuelles, rapide à implémenter et à calculer.



Pour $\alpha = 0.8$:

p	cases à p clés
0	449328
1	359463
2	143785
3	38342
4	7668
5	1226
6	163
7	18
8	1

FIGURE 40 – À gauche, probabilité qu'une case d'une table de dispersion contienne p clés chaînées. Avec un taux de remplissage de $\alpha = 0.8$, environ 45% des cases sont vides, mais seulement 1% des cases contiennent 4 éléments ou plus. À droite, la répartition prédite de 800.000 clés dans un tableau de taille 1.000.000.

Généralement la première chose à faire est de trouver une injection de $\mathcal{K}$ vers $\mathbb{N}$: ainsi il suffit de trouver un fonction de dispersion sur $\mathbb{N}$. Par exemple, pour les chaînes de caractères, nous pouvons utiliser le code ASCII de chaque caractère et interpréter la chaîne comme un nombre écrit en base 128 (ou 256). Combiné avec la règle de Horner, il s'agit donc de calculer :

$$\mathsf{code}(c[0]) + 128 \times (\mathsf{code}(c[1]) + 128 \times (\mathsf{code}(c[2]) + \ldots 128 \times \mathsf{code}(c[l-1]) \ldots))$$

Nous supposons donc maintenant que $\mathcal{K} = \mathbb{N}$. Puisque nous souhaitons une fonction de $\mathbb{N} \mapsto [0, n-1]$, une possibilité très simple est $k \mapsto k \mod n$. Combiner avec l'astuce précédente pour les chaînes de caractères, il faudra éviter de choisir $n = 128^m$, puisque alors l'indice sera le code d'un suffixe de la chaîne. Le choix de n est donc important. Cette méthode doit aussi être évité pour coder des ensembles formés de groupes d'éléments adjacents : la fonction de dispersion conserve l'adjacence, et si plusieurs de ces groupes sont envoyés vers les mêmes indices les performances vont en pâtir.

Une solution plus élaboré consiste à faire une multiplication avec une constante avant de prendre le modulo. Nous posons alors :

$$h(k) = \lfloor n \cdot [k\gamma] \rfloor$$

où $[k\gamma] = k\gamma - \lfloor k\gamma \rfloor$ est la partie fractionnaire de $k\gamma$. Il semblerait qu'en pratique, $\gamma = \frac{\sqrt{5}-1}{2}$ soit un bon choix.

Parfois, comme dans la section suivante, il est nécessaire d'avoir toute une famille de fonctions de dispersion, relativement distinctes les unes des autres. Soit p un nombre premier telle que toute clé est inférieure

à p, nous posons pour tout $a \in [1, p-1]$ et $b \in [0, p-1]$:

$$h_{a,b}(k) = ((ak + b) \mod p) \mod n$$

Ceci définit une famille de fonctions de dispersion contenant $p(p-1)$ fonctions. De plus, pour une paire de clé donnée, le nombre de ces fonctions qui produisent une collision sur ces deux clés est garanti d'être faible.

3.3 La technique du coucou

Pour certaines applications, une table de dispersion est d'abord construite, puis un très grand nombre de tests d'appartenance sont effectués, mais la table elle-même n'est plus modifiée. Nous souhaiterions dans ce cas adapter les tables de dispersion pour avoir un temps de calcul de l'appartenance aussi court que possible. C'est l'objectif de la méthode du coucou. Le but est de n'avoir qu'un seul élément dans chaque case du tableau. En cas de collision, la nouvelle clé va chasser l'ancienne. C'est de là que provient le nom de cette méthode, qui fait référence aux habitudes de cet oiseau de nos contrées, qui pond ces œufs dans le nid des autres espèces, en prenant soin de préalablement manger les œufs déjà présents. L'oisillon lui-même chasse hors du nid les éventuels œufs restants, juste après son éclosion.

Nous ne pouvons pas juste jeter l'ancienne clé chassée par l'arrivée d'une nouvelle, la parade consiste alors à insérer la clé chassée dans une deuxième table de dispersion. Une autre clé peut alors se retrouver chassée à son tour, et nous la réinsérons dans la première table de dispersion. La procédure se termine lorsque finalement une clé arrive dans une case libre, ou bien après suffisamment de tentative (dans ce cas, l'insertion échoue).

Nous utilisons donc deux tables de dispersion T_1 et T_2, avec deux fonctions de dispersions distinctes et indépendantes h_1 et h_2. Toute clé c doit impérativement être insérée, au choix :

— ou bien dans T_1 à l'indice $h_1(c)$,

— ou bien dans T_2 à l'indice $h_2(c)$.

Ainsi, tester l'appartenance d'une clé c devient extrêmement simple, avec un temps $O(1)$ dans le pire des cas, et très faible en pratique.

L'insertion se décompose en deux sous-fonctions mutuellement récursive : insère_t1 insère la clé c dans T_1, et si T_1 contenait une clé c', appelle insère_t2 pour insérer c'. insère_t2 est symétrique, elle insère la clé dans T_2, puis éventuellement appelle insère_t1 pour insérer la clé chassée. De plus, nous utilisons un compteur gardant le nombre d'appels effectués : s'il devient trop grand, l'insertion échoue. Nous verrons plus loin comment gérer ces échecs. Le code est donné en Figure 41.

Exemple 3.3. Nous construisons une table de dispersion pour l'ensemble de chaînes de caractères suivant : {“Joyeux”, “Atchoum”, “Prof”, “Simplet”, “Grincheux”, “Timide”, “Dormeur”}. Nous prenons T_1 et T_2 de taille 7. Nous utilisons la règle de Horner avec le code ASCII, en faisant les opérations modulo $p = 9949$. Nous choisissons deux fonctions de dispersion dans la famille que nous avons introduite précédemment :

$$h_1(x) := ((1385x + 5965) \mod p) \mod 7$$

$$h_2(x) := ((9838x + 4983) \mod p) \mod 7$$

Les valeurs obtenus sont donnés dans la Figure 42. Nous insérons ensuite chaque élément dans les tables T_1 et T_2 selon la méthode du coucou. Une clé k_0 est toujours initialement insérée dans T_1. Si une clé k_1 est présente dans la case d'insertion, elle est chassée vers T_2. Si à son tour la case de T_2 d'indice $h_2(k_1)$ contient

```
1   fonction appartient(entier clé, tableau d'entier t1, tableau d'entier t2) : bool =
2     (t1[h1(clé)] = clé) ∨ (t2[h2(clé)] = clé)
3
4   fonction insère_t1(entier clé, tableau d'entier t1, tableau d'entier t2, entier compteur) : void =
5     si compteur > max_iteration alors ECHEC
6     sinon
7       soit clé_chassée := t1(h1(clé))
8       t1(h1(clé)) ← clé
9       si clé_chassée ≠ ⊥ alors
10        insère_t2(clé_chassée, t1, t2, compteur)
11
12  fonction insère_t2(entier clé, tableau d'entier t1, tableau d'entier t2, entier compteur) : void =
13    soit clé_chassée := t2(h2(clé))
14    t2(h2(clé)) ← clé
15    si clé_chassée ≠ ⊥ alors
16      insère_t1(clé_chassée, t1, t2, compteur + 1)
17
18  fonction insère(entier clé, tableau d'entier t1, tableau d'entier t2) : void =
19    insère_t1(clé, t1, t2, 0)
```

FIGURE 41 – Pseudocode pour la recherche et l'insertion dans une table de dispersion avec la méthode coucou

une clé k_2, k_2 est chassée dans T_1, etc. Dans cet exemple, "Joyeux" chasse "Atchoum" lors de la deuxième insertion. "Timide" chasse "Simplet" lors de la sixième insertion. Enfin lors de la dernière insertion, plusieurs clés sont chassées successivement, jusqu'à l'insertion d'"Atchoum" dans T_2.

Il est possible que la procédure d'insertion se mette à cycler : plusieurs clés se chassent les unes les autres sans fin. Nous modélisons le problème par un graphe :

— l'ensemble des sommets est l'ensemble des cases de T_1 et T_2,
— l'ensemble des arêtes est celui des clés,
— une clé a pour extrémité les deux cases où elle peut être insérée.

Pour que l'algorithme d'insertion cycle, il est clairement nécessaire que le graphe lui-même possède un cycle : chaque insertion élémentaire concerne une arête, et deux arêtes sont considérées consécutivement seulement si elles sont incidentes à un sommet commun. Si l'insertion ne termine pas, certaines arêtes vont être visitées plusieurs fois, il existe donc un cycle dans le graphe. (Ce n'est en revanche pas une condition nécessaire, l'algorithme peut terminer même en présence d'un cycle dans certains cas.)

Pour borner la probabilité que l'insertion ne termine pas pendant une séquence de n insertions, nous bornons donc la probabilité (légèrement plus grande) qu'un cycle se forme. Il est important pour pouvoir obtenir cette borne que la taille des deux tableaux soit choisie suffisamment grande, pour l'instant nous posons $\beta = \frac{n}{m}$, où n sera le nombre maximum de clés, m la taille de chacun des deux tableaux. Le taux de remplissage de la table est alors $\beta/2$. En plus du paramètre β, il faut choisir une valeur pour max_iteration, le nombre maximum de clés pouvant être chassées lors d'une seule insertion. Des analyses assez fines, confirmées par l'expérimentation, montrent que β doit être pris inférieur à 1, et max_iteration doit être de l'ordre de $C \log n$ ($C = 5$, $\beta = 0.9$ semble convenir).

Nous nous contentons d'une analyse un peu moins bonne.

clé	valeur	h_1	h_2
“Joyeux”	7548	4	1
“Atchoum”	8287	4	0
“Prof”	3898	1	0
“Simplet”	2729	5	2
“Grincheux”	1786	6	5
“Timide”	571	5	1
“Dormeur”	2702	5	3

	T_1	T_2
0		
1	“Prof”	“Joyeux”
2		“Simplet”
3		
4	“Atchoum”	
5	“Timide”	
6	“Grincheux”	

	T_1	T_2
0		“Atchoum”
1	“Prof”	“Timide”
2		“Simplet”
3		
4	“Joyeux”	
5	“Dormeur”	
6	“Grincheux”	

FIGURE 42 – En haut, les clés et leurs indices de dispersions selon les fonctions de dispersion des tables T_1 et T_2. En bas à gauche, la table après insertion des 6 premières clés (insérées de haut en bas) : “Joyeux” a été chassé par “Atchoum”, “Simplet” a été chassé par “Timide”. Enfin à droite, le résultat de l'insertion de “Dormeur” : il chasse “Timide” dans T_2, qui chasse “Joyeux” dans T_1, qui chasse à son tour “Atchoum” dans T_2, qui trouve une place libre.

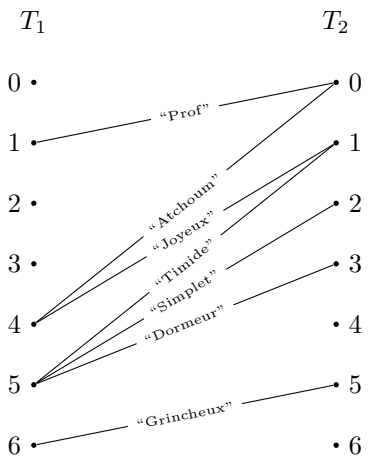


FIGURE 43 – Le graphe associé à la table de dispersion de la Figure 42.

Exemple 3.4. La Figure 43 représente le graphe dérivé de l'exemple de la Figure 42.

Lemme 3.3. *Soit G le graphe associé à une double table de dispersion constituée de deux tableaux de taille m, alors la probabilité qu'une séquence de $n < m$ insertions échoue est au plus $\frac{\beta^2}{2(1-\beta^2)} + O\left(\frac{1}{m}\right)$, avec $\beta = \frac{n}{m}$.*

Démonstration. En premier lieu, chaque arête reliant une case de T_1 et une case de T_2, il y a m^2 possibilités pour les extrêmités d'une arête. Puisque nous supposons nos fonctions de dispersion parfaites, toutes ces possibilités sont équiprobables. Si le nombre de clés présentes est k, chaque paire de case $(T_1[i], T_2[j])$ a une probabilité $\frac{k}{m^2}$ d'être présente dans G.

Nous bornons maintenant la probabilité P_l qu'il existe un chemin de longueur $2l+1$ entre deux cases données de T_1 et T_2 respectivement, en présence de k clés, par récurrence sur l. Nous montrons pour tout $i, j \leq m$:

$$P_l(i,j) := \Pr[\text{il existe un chemin de } T_1[i] \text{ vers } T_2[j] \text{ de longueur } 2l+1] \leq \frac{k^{2l+1}}{m^{2l+2}}$$

Pour $l = 0$, il existe un chemin entre $T_1[i]$ et $T_2[j]$ de longueur 1 si et seulement si il existe une arête entre ces deux cases, et cette probabilité est donc bornées par $\frac{k}{m^2}$:

$$P_0(i,j) \leq \frac{1}{n}$$

Notons $i \to^l j$ l'événement "il existe un chemin de $T_1[i]$ vers $T_2[j]$ de longueur l", et $i \to j$ pour $i \to^1 j$.

Pour $l \geq 1$, supposons notre hypothèse vrai pour toute longueur plus petite que l. Alors

$$\begin{aligned}
P_l(i,j) &\leq \Pr[\exists j', i' \ : \ i \to j' \wedge j' \to i' \wedge i' \to^{2l-1} j'] \\
&\leq \sum_{i',j'} \Pr[i \to j' \wedge j' \to i' \wedge i' \to^{2l-1} j'] \\
&\quad \text{(parce que } \Pr[A \vee B] \leq \Pr[A] + \Pr[B]) \\
&\leq \sum_{i',j'} \Pr[i \to j'] \cdot \Pr[j' \to i'] \cdot P_{l-1}(i', j) \\
&\quad \text{(parce que } \Pr[A \wedge B] = \Pr[A] \cdot \Pr[B|A], \text{ mais ici } \Pr[B|A] \leq \Pr[B]) \\
&\leq \sum_{i',j'} \frac{k}{m^2} \cdot \frac{k}{m^2} \cdot \frac{k^{2(l-1)+1}}{m^{2(l-1)+2}} \\
&\quad \text{(par hypothèse de récurrence)} \\
&= \frac{k^{2l+1}}{m^{2l+2}} \\
&\quad \text{(car il y a } m \text{ choix pour } i' \text{ et autant pour } j')
\end{aligned}$$

Nous pouvons maintenant borner la probabilité qu'il existe un chemin de longueur quelconque :

$$\Pr[\exists l \ : \ i \to^l j] \leq \sum_{l \geq 0} P_l(i,j) \leq \frac{k}{m^2} \sum_{l \geq 0} \frac{k^{2l}}{m^{2l}} \leq \frac{k}{m^2} \cdot \frac{1}{1 - \left(\frac{k}{m}\right)^2}$$

La séquence d'insertions échoue si au moins une des insertions échoue, nous bornons alors cette probabilité par celle qu'il existe k tel qu'il se forme un cycle lors de l'insertion de la k^{e} clé :

$$\begin{aligned}
\sum_{k=1}^{n} \frac{k}{m^2} \cdot \frac{1}{1 - \left(\frac{k}{m}\right)^2} &= \frac{1}{m^2(1-\beta^2)} \sum_{k=1}^{n} k \\
&= \frac{n^2}{2m^2(1-\beta^2)} + O\left(\frac{n}{m^2}\right) \\
&= \frac{\beta^2}{2(1-\beta^2)} + O\left(\frac{1}{m}\right)
\end{aligned}$$

ce qui termine cette preuve. □

En prenant $\beta < \sqrt{\frac{2}{3}} \sim 0.8$, nous obtenons une probabilité d'échec strictement inférieure à 1. En fait il s'agit d'une analyse peu précise, et même des valeurs de β plus proche de 1 (mais toujours inférieures) donnent des probabilités suffisamment faibles pour des applications pratiques.

Un autre enseignement de cette preuve est que l'insertion d'un élément dans une table contenant n clés échoue avec probabilité au plus $\Pr[\exists l \ : \ i \to^j l] \leq \frac{n^3}{m^4} \cdot \frac{1}{1-\left(\frac{n}{m}\right)^2} = O\left(\frac{1}{n}\right)$. Lorsqu'une insertion échoue, nous tirons deux nouvelles fonctions de dispersions et créons un nouvelle table de dispersion avec ces deux fonctions. En raffinant nos analyses, il est possible de montrer qu'alors l'espérance du temps d'insertion est constant : pour cela il suffirait de montrer qu'il est constant s'il n'y a pas d'échec, ce que nous pourrions

démontrer en utilisant la borne sur les longueurs de chemins. Comme la probabilité d'échec est $O\left(\frac{1}{n}\right)$, et que reconstruire une table prend un temps $O(n)$ et échoue avec une probabilité constante p, nous obtenons en espérance $O(1) + \frac{1}{n} \cdot \frac{1}{1-p} \cdot O(n) = O(1)$, où $\frac{1}{1-p}$ est le nombre moyen d'échec de la reconstruction de la table.

3.4 Comparaison avec les arbres binaires de recherche

Nous concluons cette partie en rappelant les avantages et les points faibles des deux approches que nous avons vues pour implanter des dictionnaires.

- les tables de dispersions nécessitent des fonctions de dispersion, les arbres de recherche n'ont besoin que d'une fonction de comparaison, ce qui est en général moins contraignant.
- pour la complexité des opérations élémentaires, les tables de dispersion sont clairement avantageuses ($O(1)$ contre $O(\log n)$).
- les tables de dispersions demandent une bonne calibration pour atteindre tout leur potentiel (mais sont en général déjà relativement performantes sans paramêtrer). Les AVL-arbres fonctionnent directement. Les B-arbres demandent un bon choix pour B.
- Dans les arbres binaires de recherche, les éléments sont triés. Il est donc possible de rajouter plusieurs fonctions, pour retourner par exemple le minimum, le maximum, la liste des éléments triés ou bien encore le i^{e} plus petit élément en temps $O(n)$. Dans les tables de dispersions, les éléments sont placés de façon la plus désordonnée possible, donc ces fonctions ne peuvent s'écrire.
- les arbres binaires de recherche peuvent être codés de façon persistente : les opérations ne détruisent pas l'arbre mais en créent un nouveau (et qui partage des sous-arbres avec le précédent), ainsi chaque version de l'arbre peut être gardée en mémoire. Les tables de dispersion reposent sur des tableaux, et n'ont donc pas cette propriété que certains algorithmes exploitent.

Il convient donc lors de l'implantation d'un dictionnaire de choisir laquelle des deux structures semblent la plus adaptée pour la tâche à accomplir.

4 Algorithmique des graphes

4.1 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 des 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 a certains employés seulement ?

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

4.1.1 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 4.1. *Une* graphe orienté *est un couple (V, E) de deux ensembles (que nous supposerons toujours finis), muni de deux fonctions $src : E \to V$ et $dst : E \to V$. Les éléments de V sont appelés* sommets *ou parfois* nœuds*, ceux de E sont appelés* arcs*. Les fonctions src et dst associe 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 44 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.

Définition 4.2. *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 $N(u)$.*

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

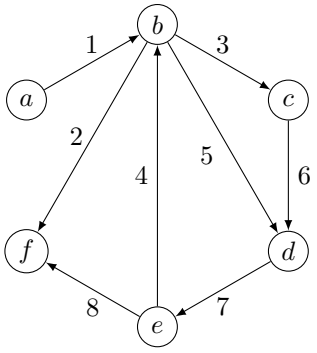


FIGURE 44 – Un exemple de représentation d'un graphe orienté, $V = \{a, b, c, d, e, f\}$, $E = \{1, 2, 3, 4, 5, 6, 7, 8\}$

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

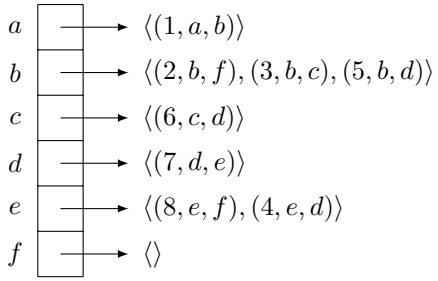
On notera systématiquement, sauf si cela devait créer confusion, par n le nombre de sommets du graphe dont il est question, et par m le nombre de ses arcs. La complexité de algorithmes que nous donnerons sera calculée asymptotiquement par rapport à ces deux paramêtres. En général, les graphes que nous 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.

4.1.2 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 45, 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 4.1. Le graphe de la Figure 44 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 $O(n)$).

4.1.3 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 4.2. Le graphe de la Figure 44 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.

4.2 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 4.3. *Un* chemin *dans un graphe (V, E) est une séquence d'arcs $P = e_1, \ldots, e_l$ 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 4.4. *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.

4.2.1 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, $(\text{src} \circ \text{pred})^l(u), (\text{src} \circ \text{pred})^{l-1}(u), \ldots, \text{src}(\text{pred}(u)), u$ 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.

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 47. 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 46.

La Figure 48 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 4.1. *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 de l'algorithme 46 est bien accessible depuis la source $s = [\![\mathsf{racine}]\!]$. 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 sont prédecesseur est défini en ligne 13. Dans ce second cas, le prédecesseur est un arc $a = [\![\mathsf{a}]\!]$ de destination v, par la définition de $\mathsf{tête}(\mathsf{a})$. Enfin, l'arc $a = [\![\mathsf{a}]\!]$ 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

```
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.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 46 – 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).

type *collection* **de** t	référence à un multiensemble fini d'éléments de type t
collection_vide() : *collection*	$[\![\text{collection\_vide}()]\!] = \text{ref}\emptyset$
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 47 – La structure de données abstraite *insertion-extraction*

par l'algorithme.

Soit $i < k$, supposons donc que v_i est parcouru. Lors de l'itération pendant laquelle $v_i = [\![\mathsf{v}]\!]$, c'est-à-dire lorsque v_i est marqué comme parcouru, e_{i+1} est un arc sortant de $[\![\mathsf{v}]\!]$, 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 4.2. *L'algorithme 46 sur un graphe G et une source s termine après au plus $2|E(G)|$ opérations sur la structure S, $|E(V)|$ insertion 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 $[\![\mathsf{v}]\!] = v$), à cause du test de la ligne 11 et de l'ajout de $[\![\mathsf{v}]\!]$ 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'opération 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 4.1. *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 $O(|V(G)| + |E(G)|)$.*

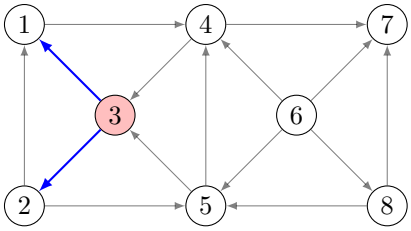
À tout parcours de graphe, on peut associer l'ordre d'entrée dans les sommets par l'ordre dans lequel les sommets sont visités :

Définition 4.5. *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.*

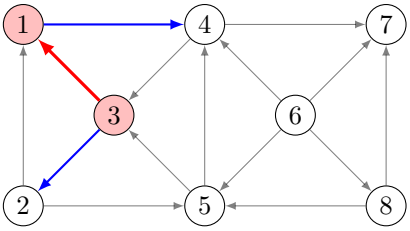
Ainsi pour l'exemple de la Figure 48, l'ordre d'entrée obtenu est $3 < 1 < 2 < 4 < 5 < 7$.

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 précedent celle pendant laquelle v est inséré, autrement dit $u < v$ dans l'ordre d'entrée. Ainsi, la suite $(\mathrm{src} \circ \mathrm{pred})^k(u)$ 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 $v_l = v$. 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

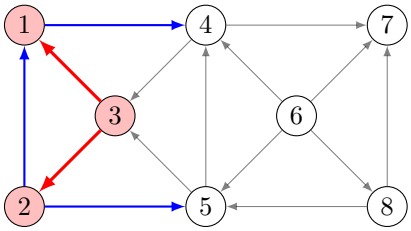
Définition 4.6. *Étant donné le parcours d'un graphe G depuis une source s, ayant produit la fonction prédecesseur pred, pour tout sommet parcouru v de G on appelle* chemin prédécesseur *le chemin $e_1, e_2, \ldots, e_l$ avec $e_i = pred(v_i)$, $v_i = src(e_{i+1})$, $v_l = v$ et $v_0 = s$. De plus, on appelle* arborescence *du parcours l'arbre enraciné en s dont les arcs sont l'image de pred.*



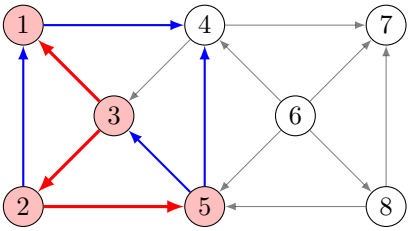
(a) Initialisation, en source 3



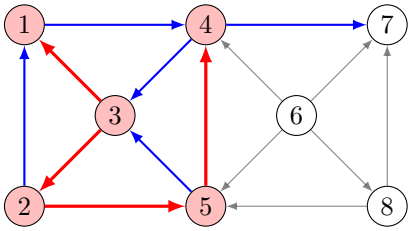
(b) avancée par (3,1) et exploration de 1



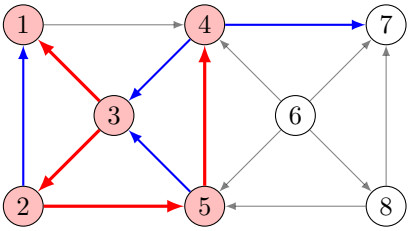
(c) avancée par (3,2) et exploration de 2



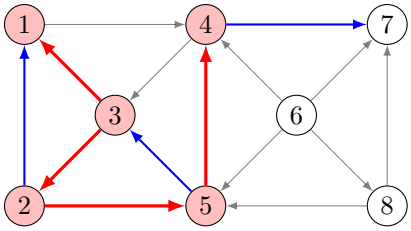
(d) avancée par (2,5), exploration de 5



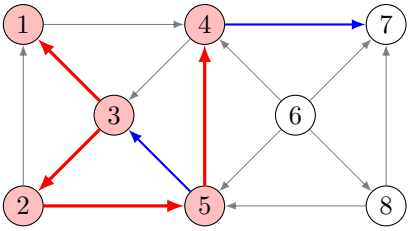
(e) avancée par (5,4) et exploration de 4



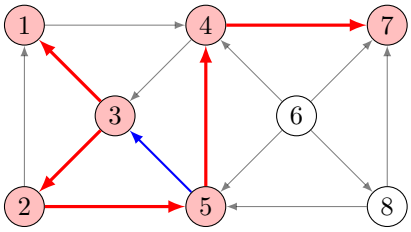
(f) avancée par (1,4), sommet 4 déjà visité



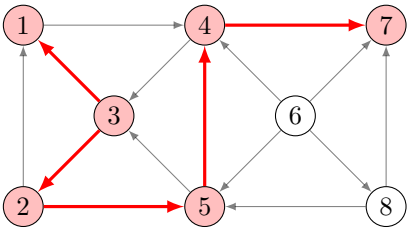
(g) avancée par (4,3), 3 déjà visité



(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 48 – 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.

4.2.2 Parcours en largeur

L'algorithme de parcours 46 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 49. 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 4.7. *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 4.3. *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.

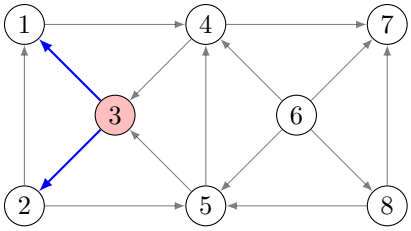
(ii) si un sommet u est parcouru, tous les sommets strictement plus proches que u de la source s ont été parcourus.

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.

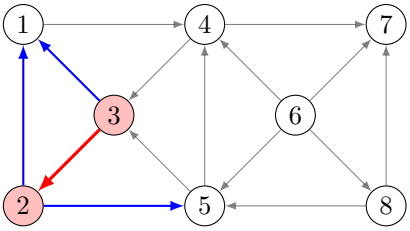
Considérons une itération quelconque et supposons l'hypothèse vraie au début de toutes les itérations précédentes et de celle-ci. Si le test de la ligne 11 est faux, il ne se passe rien et l'hypothèse reste vrai au début de l'itération suivante. Supposons donc que $[\![\mathsf{v}]\!]$ est visité pendant 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).

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 $w' = \mathrm{src}(\mathrm{pred}(w))$, 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. □

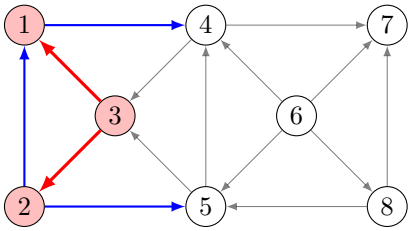
L'algorithme de parcours en largeur permet donc de calculer les plus courts chemins depuis une source. Plus exactement, il s'agit de plus courts chemins, où la longueur d'un chemin est donné par son nombre d'arcs. Nous verrons comment généraliser ce résultat lorsque chaque arc possède une longueur positive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



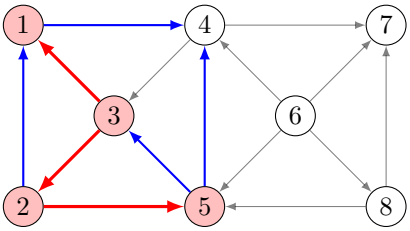
(a) Frontière : $\langle (3,1),(3,2) \rangle$



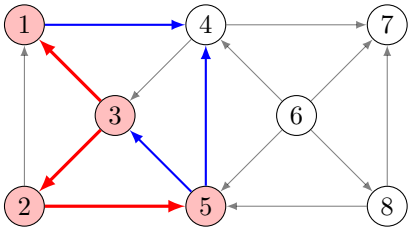
(b) Frontière : $\langle (2,1),(2,5),(3,1) \rangle$



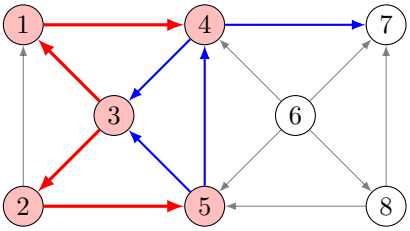
(c) Frontière : $\langle (1,4),(2,1),(2,5) \rangle$



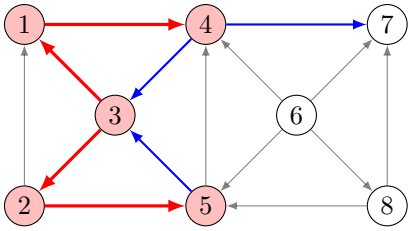
(d) Frontière : $\langle (5,3),(5,4),(1,4),(2,1) \rangle$



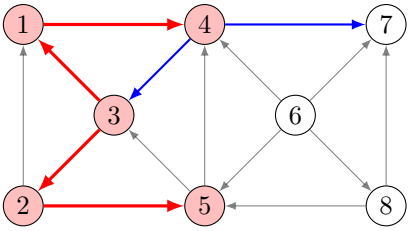
(e) Frontière : $\langle (5,3),(5,4),(1,4) \rangle$



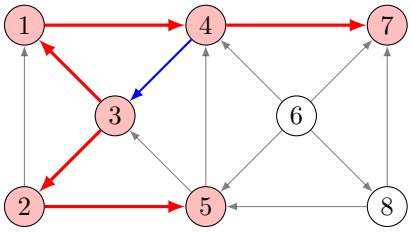
(f) Frontière : $\langle (4,3),(4,7),(5,3),(5,4) \rangle$



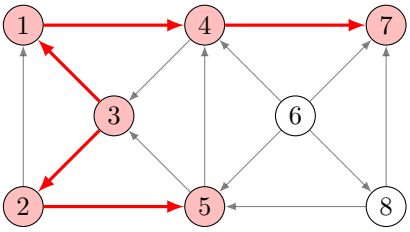
(g) Frontière : $\langle (4,3),(4,7),(5,3) \rangle$



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



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



(j) Frontière : $\langle \rangle$

FIGURE 49 – Un parcours du même graphe en largeur d'abord.

4.2.3 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 4.8. *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 extrait consécutivement, tant qu'ils sont tout 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 50.

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 51. 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 pour les algorithmes de parcours généraux, le parcours en profondeur définit un ordre d'entrée dans les sommets parcourus. Contrairement aux algorithmes généraux, pour lesquels le traitement de la visite d'un sommet n'est pas groupé temporellement : dans l'algorithme général, on ajoute tous les sommets à la frontière avant de passer au sommet suivant. 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 :

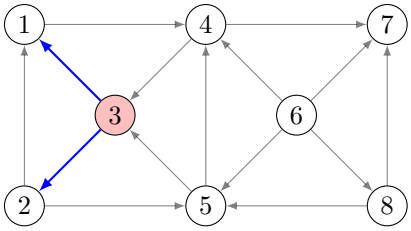
Définition 4.9. *L'ordre de sortie des sommets d'un parcours en profondeur est l'ordre total sur les sommets parcourus tel que $u < v$ si $s = u$ ou bien la fin de l'exécution de la ligne 9 lors de l'appel de* visiter *lors duquel $[\![v]\!] = u$ est parcouru précède la fin de l'exécution de la ligne 9 lors de l'appel de* visiter *lors duquel v est parcouru.*

Lemme 4.4. *À tout moment de l'exécution de l'algorithme de parcours en profondeur, Figure 51, la pile d'appels des arguments de la fonction* explore *est un sv-chemin, avec $s = [\![s]\!]$ et $v = [\![v]\!]$.*

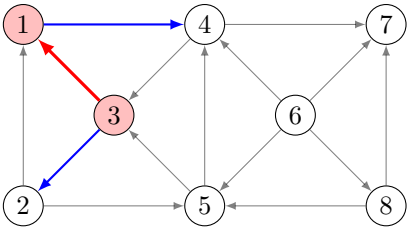
Démonstration. Lors d'un appel en ligne 10, la pile est initialement vide et on lui ajoute un arc sortant de s, donc la pile décrit bien un chemin de source s et de destination la tête de l'arc empilé. Par induction sur le nombre d'appel récursif, on analyse le cas d'un appel récursif en ligne 9. Cet appel ajoute un arc $e = [\![e]\!]$ avec $\mathrm{src}(e) = [\![v]\!]$. On obtient donc bien que la pile décrit un su-chemin avec $u = \mathrm{dst}([\![e]\!])$. □

Ceci justifie la définition suivante :

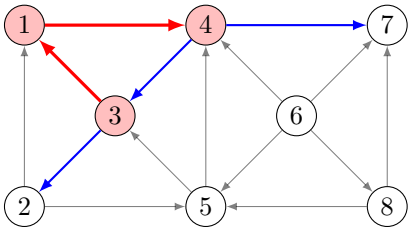
Définition 4.10. *À tout moment de l'exécution du parcours en profondeur, on appelle* chemin d'appel *le chemin décrit par la pile d'appel récursif.*



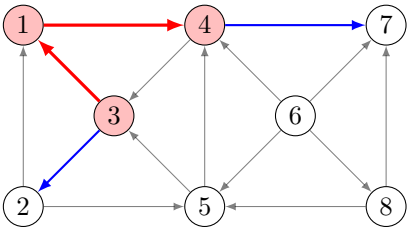
(a) Frontière : $\langle (3,1),(3,2) \rangle$



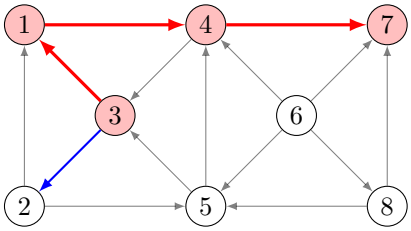
(b) Frontière : $\langle (1,4),(3,2) \rangle$



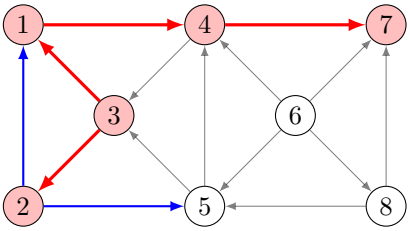
(c) Frontière : $\langle (4,3),(4,7),(3,2) \rangle$



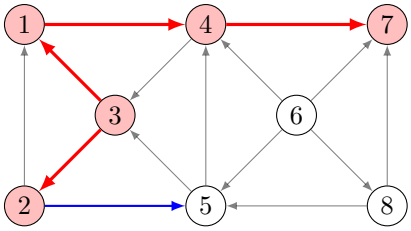
(d) Frontière : $\langle (4,7),(3,2) \rangle$



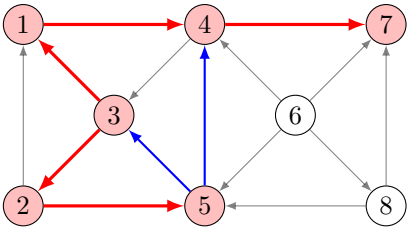
(e) Frontière : $\langle (3,2) \rangle$



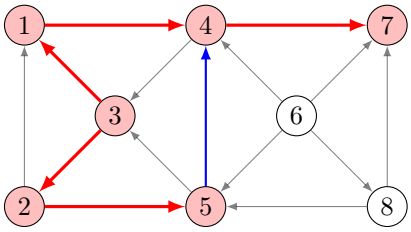
(f) Frontière : $\langle (2,1),(2,5) \rangle$



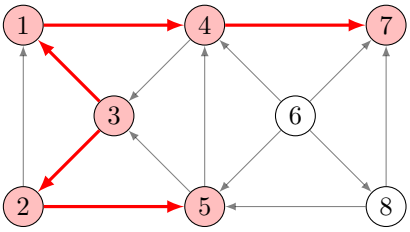
(g) Frontière : $\langle (2,5) \rangle$



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



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



(j) Frontière : $\langle \rangle$

FIGURE 50 – 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 51 – Version récursive du parcours en profondeur : cette version utilise la pile d'appels récursifs comme structure d'insertion-extraction.

Lemme 4.5. *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 51 avec $[\![v]\!] = v$).*

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'en suit. □

4.2.4 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 vont être nécessaire vont utiliser 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 52.

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 53.

4.3 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 uv, $u < v$.

```
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 52 – 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. 46.

```
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 53 – Version itérée récursive du parcours en profondeur.

Définition 4.11. *Pour un graphe orienté G, un* ordre topologique *est un ordre total $<$ sur $V(G)$ tel que pour tout arc uv, $u < v$.*

Nous commençons par chercher quand un tel ordre existe.

Définition 4.12. *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 4.6. *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 $u_0u1, u_1u_2, \ldots$. 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 $u_iu_{i+1}, u_{i+1}u_{i+2}, \ldots, u_{j-1}u_j$ 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 ramener 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 4.1. *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 $u_0u_1, u_1u_2, \ldots, u_{l-1}u_l$, 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 4.5, u_{l-1} est découvert alors que u_0 est sur le chemin d'appel. Donc explore(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 4.2. *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 4.5, $v <_s u$ comme il se doit.

Dans l'autre cas, $v <_e u$. Par contradiction, supposons que $u <_s v$. De nouveau par le Lemme 4.5, 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. □

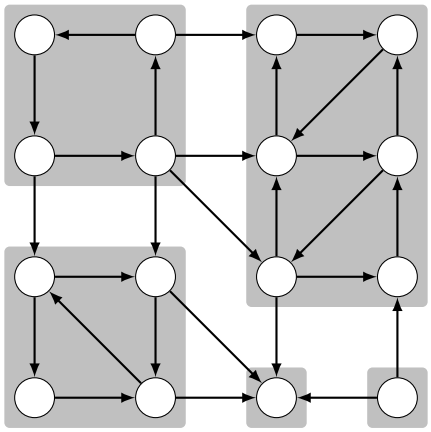
Là encore, le parcours en profondeur modifié pour attribuer un ordre de sortie (ou bien en numérotant les sommets, ou bien en retournant une liste de sommets ordonnées comme cela sera utile pour la section suivante) garde sa complexité de $\Theta(|V(G)| + |E(G)|)$.

4.4 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 4.13. *$u \in V(G)$ 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.

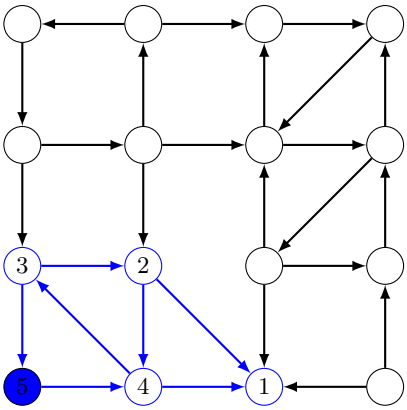
- Nous procédons à un premier parcours itéré en profondeur de G, ceci nous donne un ordre de sortie sur les sommets. Il s'agit donc du même algorithme que celui permettant de trouver un tri topologique, mais que nous utilisons sur un graphe qui n'est pas nécessairement acyclique.
- Dans cet ordre en 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 54 montre un exemple d'exécution de l'algorithme de Kosaraju.

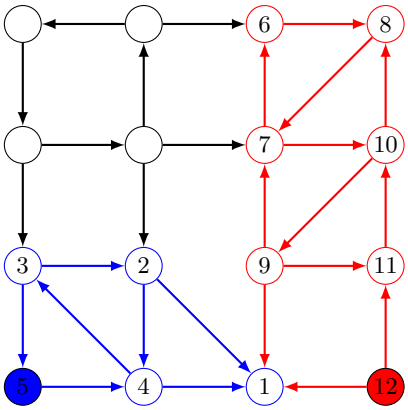
Une clé pour comprendre l'algorithme est le lemme suivant :

Lemme 4.7. *Les graphes G et $\overleftarrow{G}$ ont les mêmes composantes fortement connexes.*

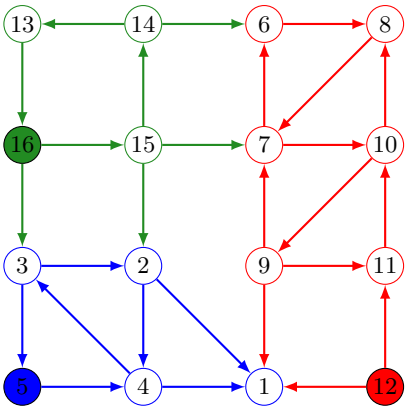
Démonstration. Un uv-chemin dans G donne lieu à un vu-chemin dans $\overleftarrow{G}$, et inversement. Donc deux sommets sont mutuellement accessibles dans G ssi ils le sont dans $\overleftarrow{G}$. □



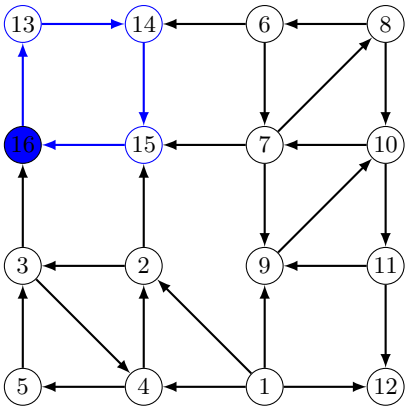
Parcours en profondeur du sommet bleu



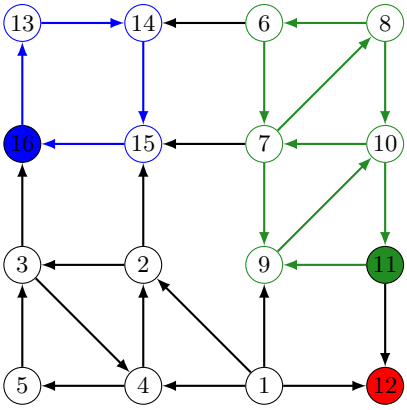
On repart du sommet rouge



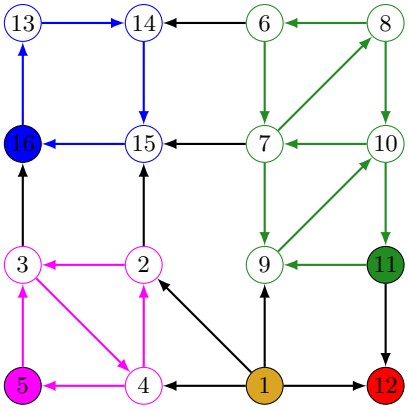
On repart du vert, ceci termine la phase 1



Première composante fortement connexe



Deuxième (rouge) et troisième (verte)



Quatrième (fuschia) et Cinquième (jaune)

FIGURE 54 – Exemple d'exécution de l'algorithme

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

Démonstration. En effet, si le sommet u est découvert lors d'une itération partant d'une source s, et que v est accessible depuis u, alors u est accessible depuis s : il existe 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 accédé 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 4.9. *Lors d'un parcours itéré, l'ensemble des sommets découvert par chaque itération est l'union de une ou plusieurs composantes fortement connexes.*

Démonstration. On applique deux fois le Lemme 4.8 : si u et v sont mutuellement accessibles, ils sont découverts lors de la même itération. □

Lemme 4.10. *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 4.5, pour tout $u \in C_1 \cup C_2$, $u \leq_s s$.

Si $s \in C_2$, par le Lemme 4.5, 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 v n'est pas dans le chemin d'appel lorsque v est découvert). □

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_1C_2 est un arc de G_C s'il existe $u \in C_1$, $v \in C_2$ et $uv \in E(G)$. Alors le Lemme précédent montre que $<_C$ est un ordre topologique de G_C.

Théorème 4.3. *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 $C_1, C_2, \ldots, C_k$ trouvées 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 4.9, 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 4.10, 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. □

4.5 Recherche de plus courts chemins depuis une source

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 4.14. *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

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

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 55. 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 4.11. *Les priorités des arcs extraits de* frontière *en ligne 15, Figure 55, forment une séquence croissante.*

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 4.4. *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 4.11 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 4.2. □

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.

4.6 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 4.15. *Soit* $G = (V, E)$ *un graphe non-orienté. Pour tout* $X \subseteq V$*, Nous notons* $\delta(X) = \{uv \in E : u \in X, v \notin X\}$*, la* coupe *de* bord X.

Proposition 4.1. *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* $X \subsetneq V$*,* $Xneq$ *tel que* $\delta(X) =$.

(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 4.16. *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 4.17. *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 4.12. *[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 4.5. *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 46. 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 4.12, 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 4.2. □

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

4.7 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 allns 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 affectés 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.

4.7.1 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 4.18. *Soit G un graphe orienté, s et t deux sommets de G, et $c : E(G) \to \mathbb{R}^+$ une fonction de capacité. Un* st-flot *de G, c est une fonction $f : E(G) \to \mathbb{R}^+$ qui vérifie les deux propriétés suivantes :*

Loi de capacité *pour tout arc $e \in E(G)$, $0 \leq f(e) \leq g(e)$,*

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

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

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-flow f est l'excès en t, excès(t). 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 4.19. *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 4.13. *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. □

4.7.2 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 4.2. *La fonction $f : e \to 0$ 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, $\text{excès}(u) = 0$ (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 4.20. *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.*

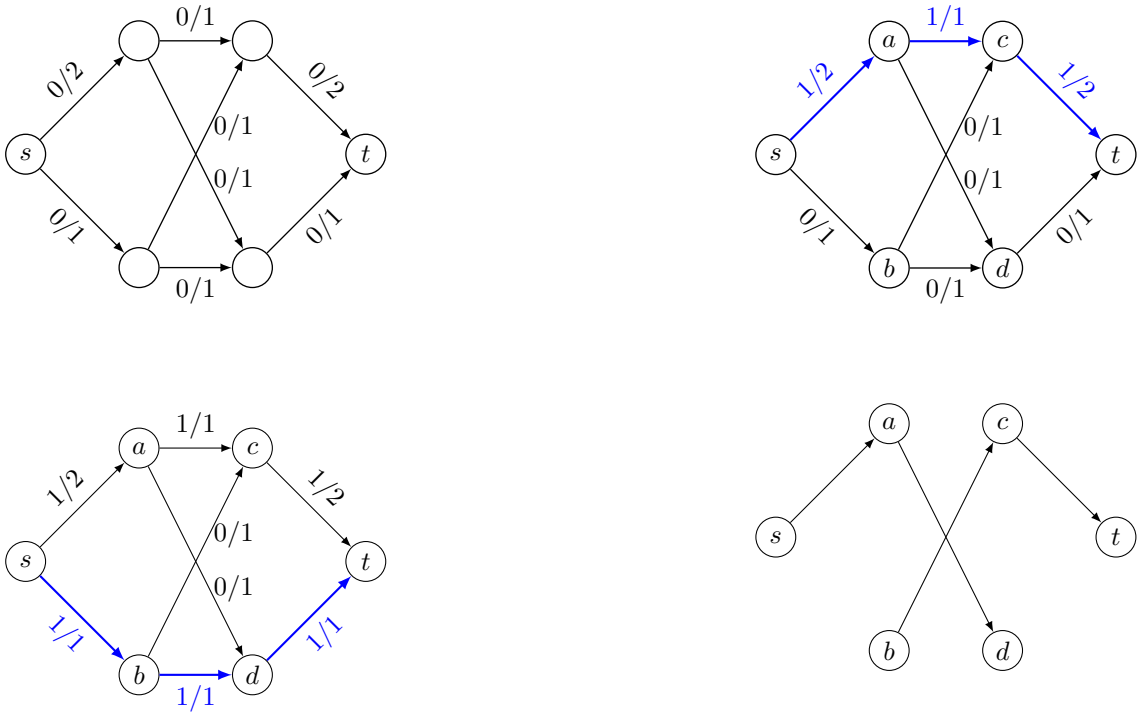


FIGURE 56 – L'algorithme naïf de calcul d'un flot : nous trouvons deux st-chemins permettant d'augmenter la valeur du flot d'une unité chacun. Puis il n'existe plus de st-chemins ne passant que par des arêtes non-saturées, comme le montre le graphe en bas à droite.

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 na pas violer la contrainte de capacité). La Figure 56 montre un exemple d'exécution de cet algorithme sur un graphe à 6 sommets. Lors de la première étape, l'algorithme choisi 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 57. 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 56. L'ensemble des sommets accessibles depuis s représentent les sommets vers lesquels il est toujours possible d'envoyer du flot. D'une certaine

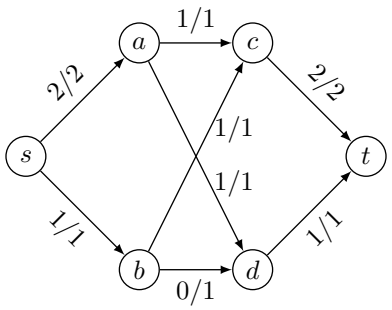


FIGURE 57 – Un flot maximum pour le graphe de la Figure 56. Il a pour valeur 3.

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

Définition 4.21. *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 arc :*

- $\overrightarrow{e}$ *pour tout $e \in E(G)$ tel que $f(e) < c(e)$, avec $src(\overrightarrow{e}) = src(e)$ et $dst(\overrightarrow{e}) = dst(e)$. e est alors un* arc avant.
- $\overleftarrow{e}$ *pour tout $e \in E(G)$ tel que $f(e) > 0$, avec $src(\overleftarrow{e}) = dst(e)$ et $dst(\overleftarrow{e}) = src(e)$. $\overleftarrow{e}$ est alors un* arc arrière.

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 définie 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 résiduelle est en effet une façon de représenter quelles sont les libertés qui nous sont donnés 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 4.14. *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 4.22. *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 56 : 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 avants, et diminuant le flot sur les arcs arrières du chemin augmentant (Figure 58).

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,
— 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 60. 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 : rappelons qu'une st-coupe 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 : $F = \delta(U)$.

Plus spécifiquement, nous avons besoin d'une coupe orienté : l'ensemble des arcs sortant d'un ensemble de sommets U, que nous avions noté $\delta^+(U) := \{uv \in E(G) \mid u \in U, v \notin U\}$.

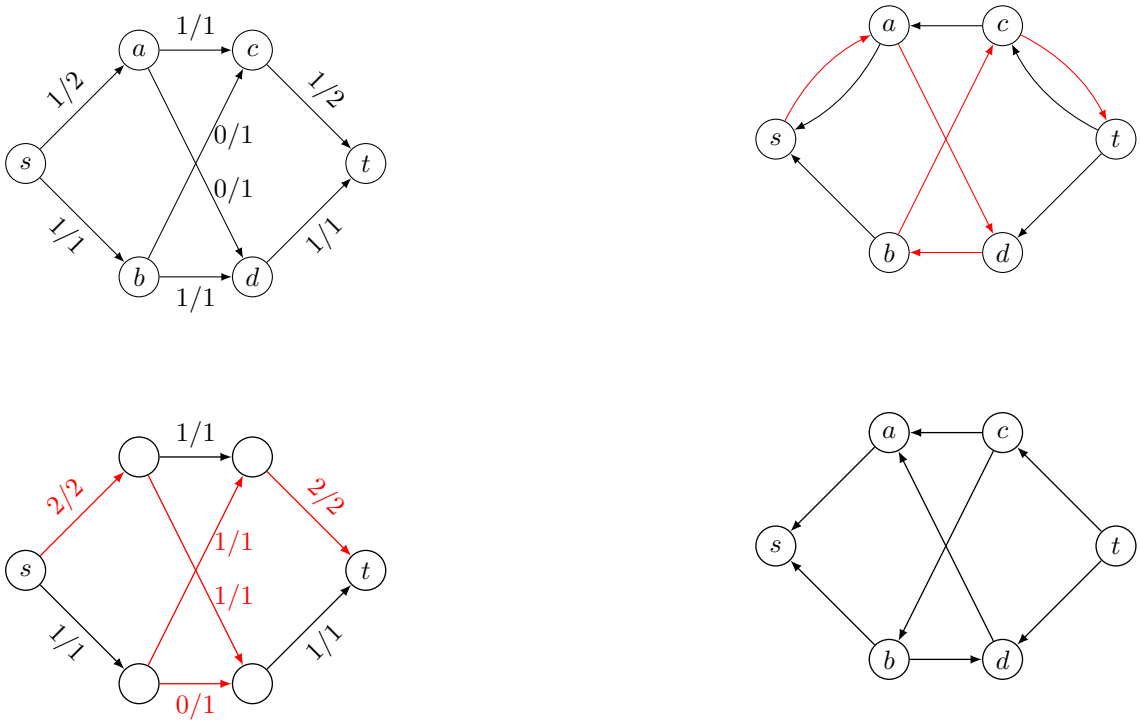


FIGURE 58 – Utilisation d'un arc arrière dans le calcul d'un chemin. Un flot sous-optimal en haut à gauche. En haut à droite le graphe résiduel pour ce flot sous-optimal. En bas à gauche le flot après augmentation, et son graphe résiduel en bas à droite. Il n'existe plus de st-chemin dans le dernier graphe résiduel, l'algorithme est terminé.

```
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 59 – 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 60 – Le cœur de l'algorithme de Ford-Fulkerson. Nous utilisons, en plus des fonctions utiles décrites en Figure 59, d'un algorithme de parcours de graphes, tel que celui en Figure 46

Lemme 4.15. *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 :

$$\begin{aligned}
\text{excès}(t) &= \sum_{v \in V(G) \setminus U} \text{excès}(v) \\
&= \sum_{v \in V(G) \setminus U} \left(\sum_{e \in \delta^-(v)} f(e) - \sum_{e \in \delta^+(v)} f(e) \right) \\
&= \sum_{e \in \delta^+(U)} f(e) - \sum_{e \in \delta^-(U)} f(e) \\
&\leq \sum_{e \in \delta^+(U)} c(e) \qquad (26)
\end{aligned}$$

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 4.6 (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 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 (26) est serrée : il y a 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 autres hypothèses, 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 4.3. *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 61 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 d'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 représentent l'essentiel des calcules 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.

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.

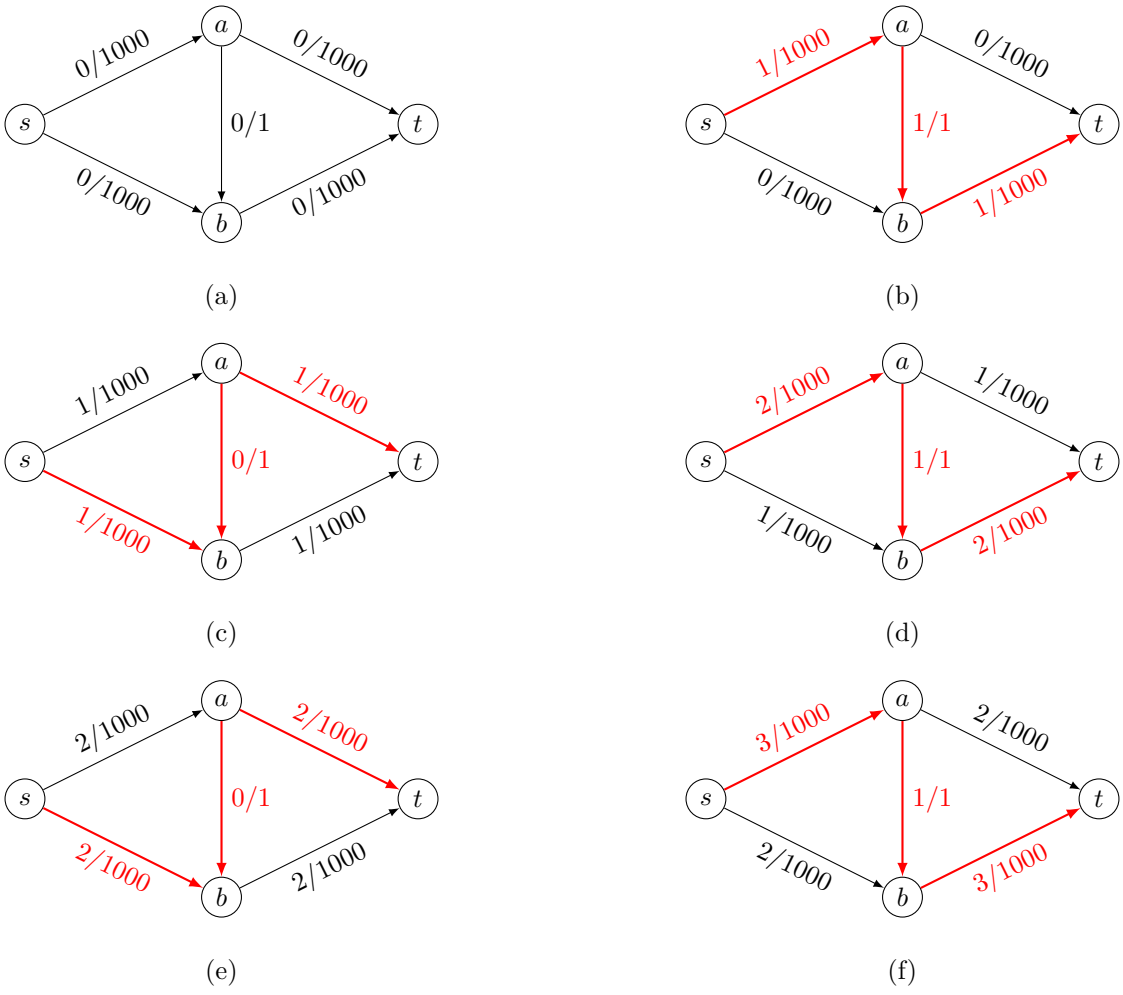


FIGURE 61 – Il faut 2000 itérations à l'algorithme pour terminer s'il choisit systématiquement un chemin passant par l'arc vertical dans cet exemple.

Lemme 4.16. *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 supprime 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 4.17. *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 4.16, 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 4.18. *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 4.16 et 4.17, 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) □

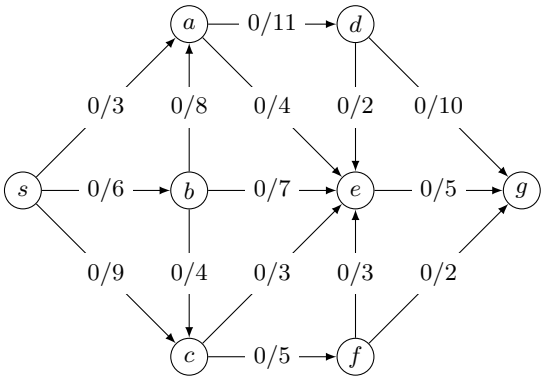
Théorème 4.7. *L'algorithme de Ford-Fulkerson peut être implémenté avec une complexité asymptotique $O(|V|.|E|^2)$.*

Démonstration. Il suffit d'ajouter au Lemme 4.18 que chaque itération a la complexité d'un parcours en profondeur, c'est-à-dire $O(|E| + |V|)$. □

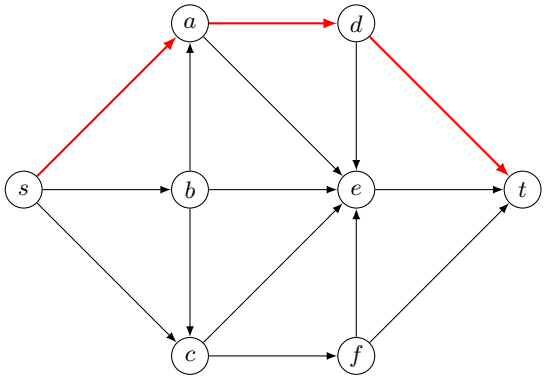
Notons que cette fois nous n'avons même pas supposé que les capacités sont entières ou rationnelles. Le choix du plus court chemin garantit une terminaison rapide de l'algorithme même en présence de très grandes capacités ou de capacités irrationnelles.

Un exemple complet Nous présentons une exécution de l'algorithme sur un exemple un peu plus gros, en Figures 62, 63 et 64. 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 larc arrière eb à deux reprises pour les chemins des graphes résiduels (j) et (k), ce qui fait baisser le flot sur l'arc be.

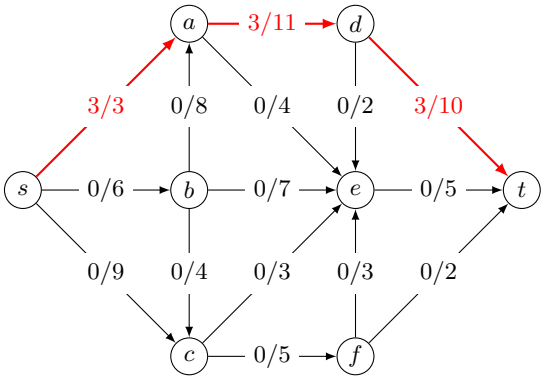
En (n), il n'existe pas de st-chemin. Ceci marque la fin de l'algorithme. Les sommets accessibles dans le dernier graphe résiduel induisent une coupe de capacité minimum. Tous les arcs sortants de cette coupe sont saturés de flot (arcs verts en (o)), les arcs entrants n'ont plus de flot (arcs pointillés en (o)). La somme des capacités des arcs verts est 16, ce qui est aussi la valeur de flot maximum, conformément au Théorème 4.6 max-flot min-coupe.



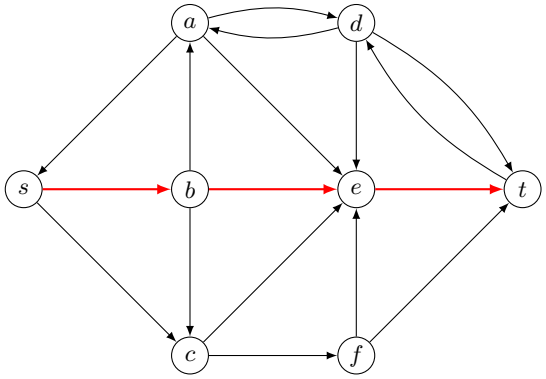
(a)



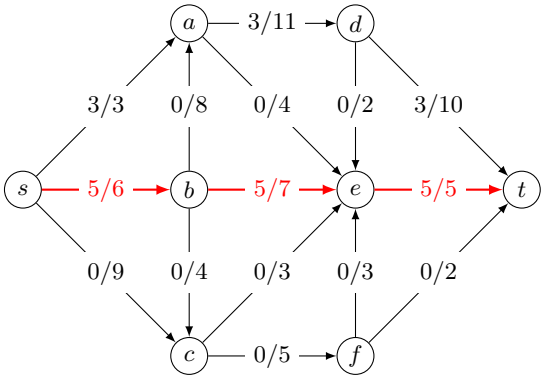
(b)



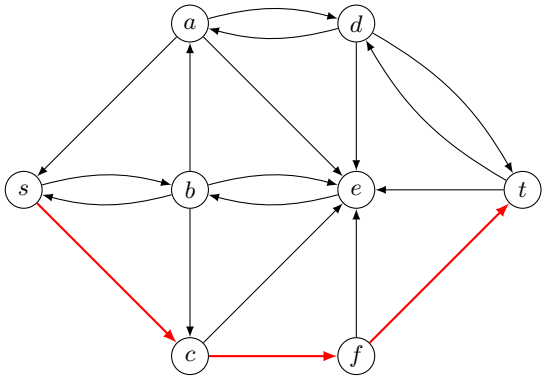
(c)



(d)

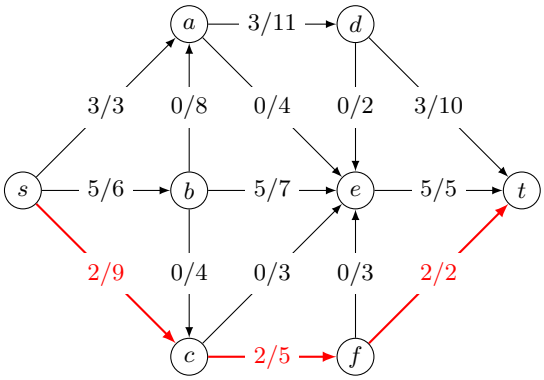


(e)

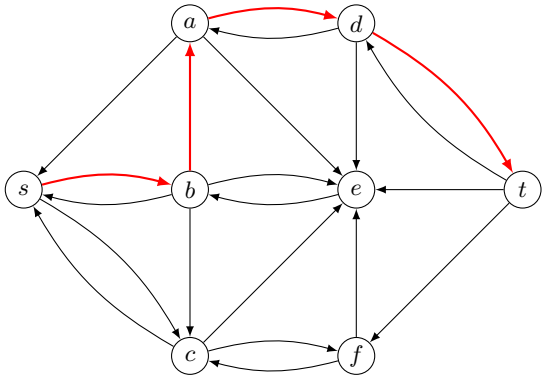


(f)

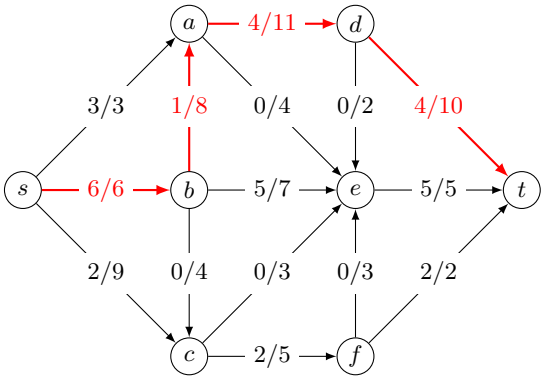
FIGURE 62 – Exécution de l'algorithme de Ford-Fulkerson, partie 1.



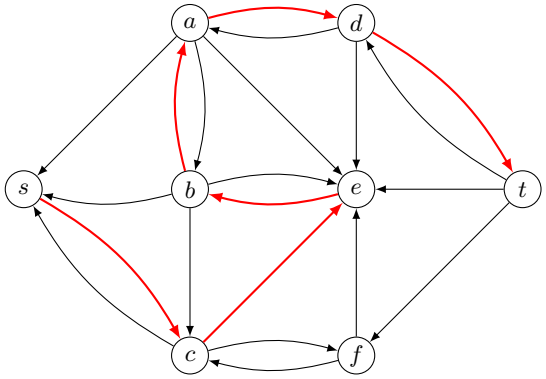
(g)



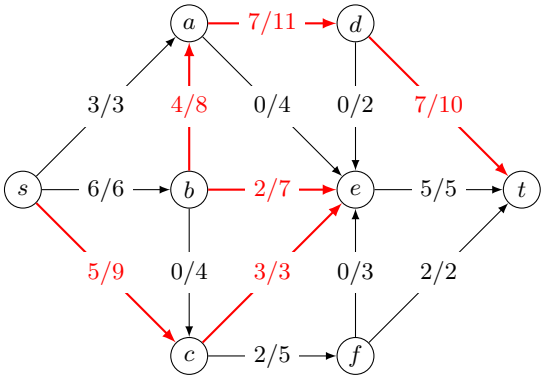
(h)



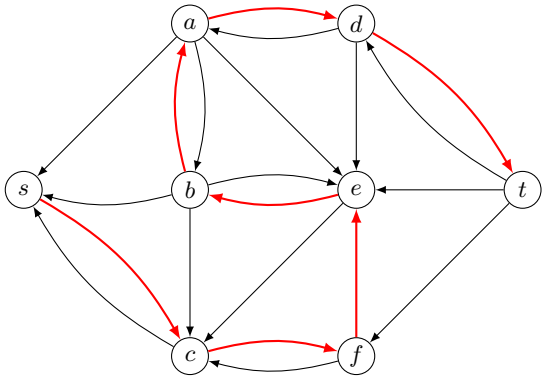
(i)



(j)

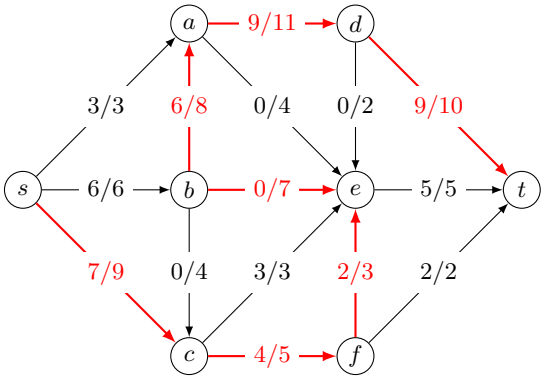


(k)

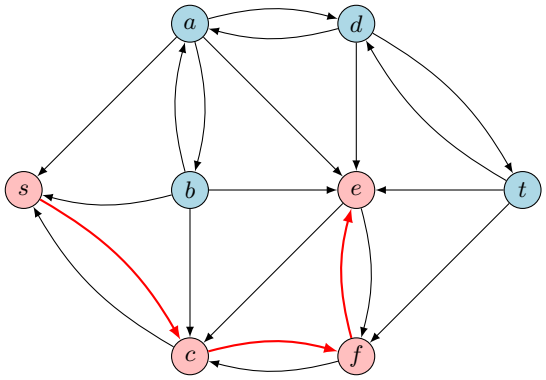


(l)

FIGURE 63 – Exécution de l'algorithme de Ford-Fulkerson, partie 2.



(m)



(n)

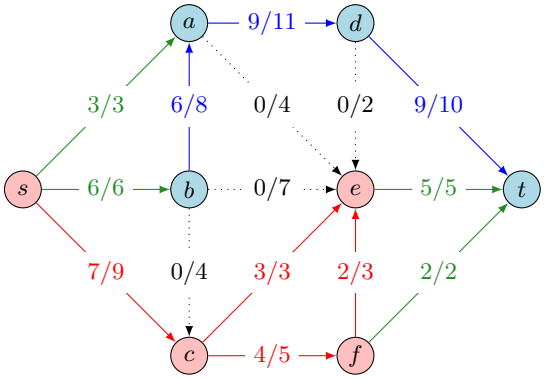


FIGURE 64 – Exécution de l'algorithme de Ford-Fulkerson, partie 3.

5 Algorithmes récursifs

5.1 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 $f : A \mapsto B$ 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 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 $1 \to 1/2 \to 1/4 \to 1/8 \to \ldots$. La définition correcte est $f(x) = 1 + f(x/2)$ si $x \geq 1$, $f(x) = 0$ sinon. Alors, si $x \geq 1$, $x - x/2$ décroit d'au moins $1/2$, ce qui nous assure que le chemin partant de x à 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 5.1. *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).*

5.2 Calcul ascendant d'une fonction récursive (*a.k.a.* 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

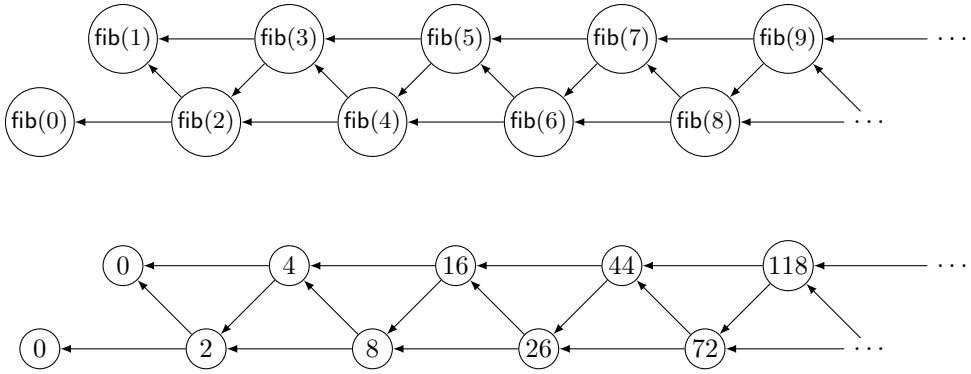


FIGURE 65 – Une représentation d'une partie du graphe G_{fib} en haut. En bas, le nombre d'appels récursifs nécessaires avec la version naïve pour le calcul d'un terme de la suite de Fibonacci.

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 provoque quatre autres, et ainsi de suite, le nombre d'appels récursifs augmentant de façon exponentielle, comme illustré par la Figure 65.

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 précédant.

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 ordre 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 technique

```
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 66 – Fibonacci mémoïsé

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 66 (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, est nécessaire.

Enfin, programmation dynamique ou mémoïsation ne sont pas toujours nécessaire, 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 aussi 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 : $\langle e_1 \rangle$, $\langle e_1, e_2 \rangle$, …, $\langle e_1, \ldots, e_{n-1} \rangle$,
— symétriquement les suffixes de la séquence : $\langle e_n \rangle$, $\langle e_{n-1}, e_n \rangle$, …, $\langle e_2, \ldots, e_n \rangle$,
— 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.

5.2.1 Exemple : plus longue sous-séquence commune

Définition 5.2. *Soit $S := \langle e_1, e_2, \dots e_n\rangle$ une séquence d'éléments d'un alphabet Σ. Une* sous-séquence *est une séquence $R \in \Sigma^\star$ de la forme $R = \langle e_{i_1}, e_{i_2}, \dots e_{i_l}\rangle$ avec $1 \leq i_1 < i_2 < \dots < i_l \leq n$. 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 donnée deux séquences S et T, nous voulons calculer la plus longue sous-séquence R apparaissant dans S et dans T.

Définition 5.3. *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 de $\triangleleft$ suivantes. Notons $a.\langle e_1, \dots, e_l\rangle$ la séquence $\langle a, e_1, \dots, e_l\rangle$, et pour un ensemble $L \subseteq \Sigma^\star$, notons $a.L := \{a.S \mid S \in L\}$.

Proposition 5.1. *Pour tous $a, b \in \Sigma$ et $S, T \in \Sigma^\star$ avec $a \neq b$:*

$$S \lhd T \implies a.S \lhd a.T \tag{27}$$
$$S \lhd T \implies S \lhd a.T \tag{28}$$
$$a.S \lhd a.T \implies S \lhd T \tag{29}$$
$$b.S \lhd a.T \implies b.S \lhd T \tag{30}$$

Démonstration. Laissée en exercice. □

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

Lemme 5.1. *Pour tous $a, b \in \Sigma$ et $S, T \in \Sigma^\star$ avec $a \neq b$,*

$$ssc(a.S, a.T) = a.ssc(S, T) \cup scc(S, T) \tag{31}$$
$$ssc(a.S, b.T) = ssc(a.S, T) \cup ssc(S, b.T) \tag{32}$$

Démonstration.

$$\begin{aligned}
\mathrm{ssc}(a.S, a.T) &= \{R \mid R \lhd a.S, R \lhd a.T\} \\
&\quad \text{(Définition)} \\
&\leq \{a.R' \mid a.R' \lhd a.S, a.R' \lhd a.T\} \cup \{b.R' \mid b \neq a, b.R' \lhd a.S, b.R' \lhd a.T\} \cup \{\varepsilon\} \\
&\quad \text{(Par distinction de la 1}^{\text{re}}\text{ lettre)} \\
&\subseteq a.\{R' \mid R' \lhd S, R' \lhd T\} \cup \{R \mid R \lhd a.S, R \lhd a.T\} \\
&\quad \text{(Équations (29) et (30))} \\
&= a.\mathrm{scc}(S, T) \cup \mathrm{scc}(S, T) \\
&\quad \text{(Définition)} \\
&\subseteq \mathrm{scc}(a.S, a.T) \\
&\quad \text{(Équation (28))}
\end{aligned}$$

Donc (31) est vraie.

$$
\begin{aligned}
\mathrm{ssc}(a+S, b+T) &= \{R \mid R \lhd a.S, R \lhd b.T\} \\
&\qquad \text{(Définition)} \\
&= \{a.R' \mid a.R' \lhd a.S, a.R' \lhd b.T\} \\
&\quad \cup \{b.R' \mid b.R' \lhd a.S, b.R' \lhd b.T\} \\
&\quad \cup \{c.R' \mid c \notin \{a,b\}, c.R' \lhd a.S, c.R' \lhd b.T\} \\
&\qquad \text{(Par distinction de la 1}^{\text{re}}\text{ lettre de } R) \\
&\subseteq \{R \mid R \lhd S, R \lhd b.T\} \cup \{R \mid R \lhd a.S, R \lhd T\} \cup \{R \mid R \lhd S, R \lhd T\} \\
&\qquad \text{(Multiples applications de l'équation (30))} \\
&= \mathrm{scc}(S, b.T) \cup \mathrm{scc}(a.S, T) \cup \mathrm{scc}(S.T) \\
&\qquad \text{(Définition)} \\
&= \mathrm{ssc}(S, b.T) \cup \mathrm{scc}(a.S, T) \\
&\qquad \text{(Car } \mathrm{scc}(S,T) \subseteq \mathrm{scc}(S, b.T) \text{ par l'équation (28))} \\
&\subseteq \mathrm{scc}(a.S, b.T) \qquad \text{(Équation (28))}
\end{aligned}
$$

Donc (32) est vraie. □

Ceci nous donne immédiatement la formule de récurrence :

Corollaire 5.1. *Pour tous $a, b \in \Sigma$, $S, T \in \Sigma^\star$ avec $a \neq b$:*

$$plscc(\varepsilon, S) = \{\varepsilon\} \tag{33}$$

$$plssc(a.S, a.T) = a.plssc(S, T) \tag{34}$$

$$plssc(a.S, b.T) = maximaux(plssc(S, b.T) \cup plssc(a.S, T)) \tag{35}$$

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 l'élément d'indice i, j représente la paire (S', T') avec $|S'| = i$ et $|T'| = j$. Le code est alors donné en Figure 67 avec un calcul ascendante de la formule récursive, et en Figure 68 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 5.2. *Les algorithmes des Figures 67 et 68 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 67 : 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 68, 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 $O(nm)$. □

```
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 67 – 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 68 – Algorithme de recherche d'une plus longue sous-séquence commune, avec mémoïsation.

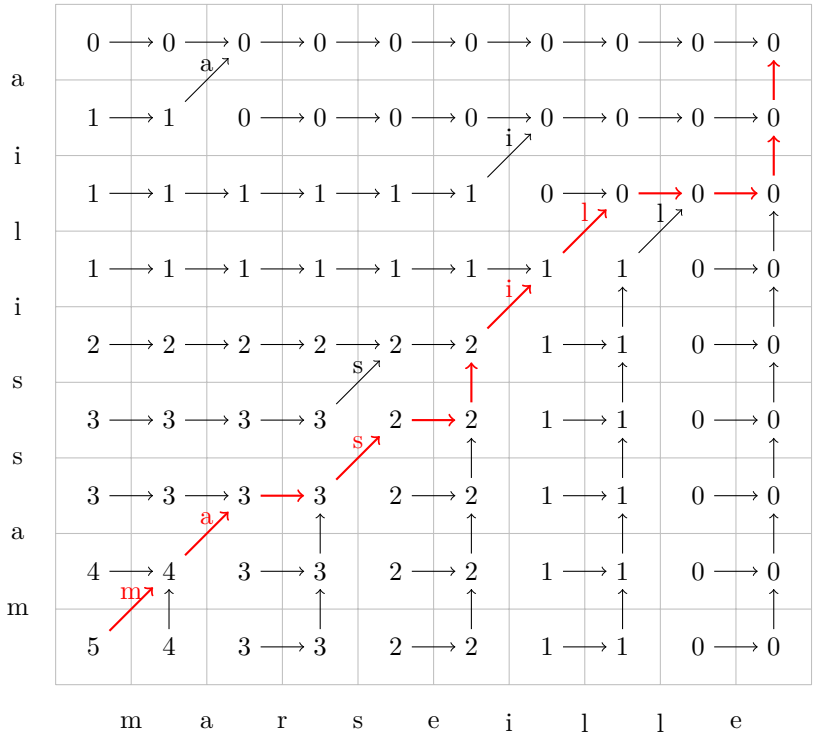


FIGURE 69 – Représentation par tableau du calcul de la plus longue sous-séquence commune des mots `marseille` et `massilia`. Chaque case correspond au sous-problème sur les suffixes formés respectivement avec les lettres des colonnes à droite et avec les lettres des lignes au-dessus. La valeur dans la 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`.

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, $\mathsf{s} = \mathsf{t}$ et plssc n'est appelé que sur les suffixes de même longueur, soit une complexité de $\Theta(n)$ 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.

5.2.2 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 constitué 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 70 présente un exemple de cette structure, qui montre comment les appariements influencent les repliements de la molécule.

Définition 5.4. *Notons $w_1 w_2 \dots w_l \in \Sigma^l$ 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 70 est ainsi repris en Figure 71.

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 optimal.

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 optimal.

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

Définition 5.5. *Pour tout mot $w \in \Sigma^\star$, notons $\nu(w)$ le nombre maximum de paires d'un appariement de w.*

Proposition 5.2. *Soit $u, v \in \Sigma^\star$, 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 $\nu(x.u.y.v) \geq 1 + \nu(u) + \nu(v)$.*

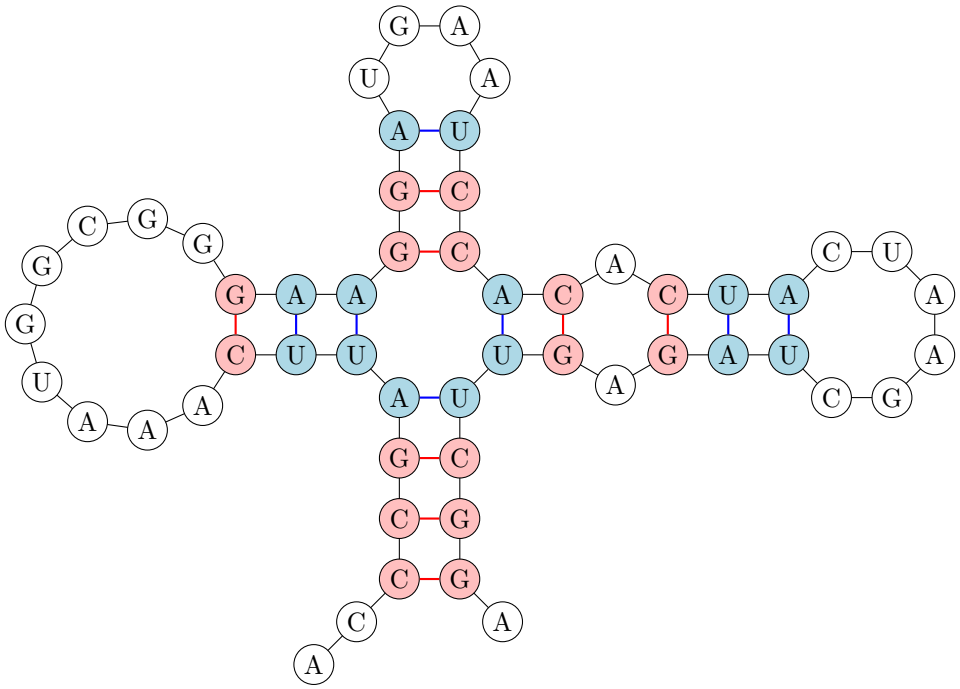


FIGURE 70 – Un exemple de structure secondaire d'un brin d'ARN.

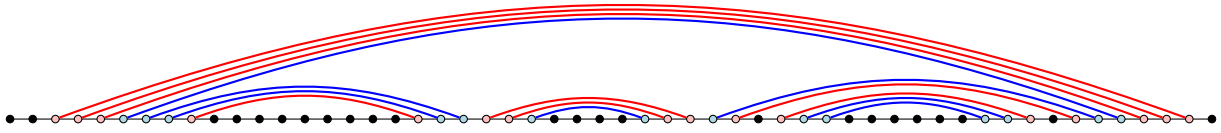


FIGURE 71 – La même structure secondaire présentée *à plat*.

Démonstration. L'union $S_{u.v}$ des appariements optimaux 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 5.3. *Pour tout mot $w = w_1 \dots w_l \in \Sigma^\star$ avec $l > 4$, et S un appariement optimal pour w.*

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

Démonstration. Premier cas, il existe j avec $\{1, j\} \in S$. Par la propriété (ii), $j > 4$.Notons $u := w_2, \dots, w_{j-1}$ et $v := w_{j+1}, \dots, w_l$, 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 optimal S, S induit un appariement sur $w_2 \dots w_l$, donc $nu(w_2 \dots w_l) \geq \nu(w) \geq \nu(w_2 \dots w_l)$. □

Le Lemme 5.3 nous fournit une formule de récurrence pour le calcul de l'appariement optimal. 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 chaines 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 5.3 pour le choix de j. Nous en déduisons la complexité de l'algorithme obtenu, présenté en Figure 72.

Lemme 5.4. *La complexité de l'algorithme de la Figure 72 pour le calcul d'un appariement optimal d'une séquence d'ARN de longueur n est $O(n^3)$ asymptotiquement dans le pire des cas.*

L'agorithme décrit en Figure 72 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 chaines, 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

5.2.3 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 recherché 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 :

```
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 72 – Algorithme d'appariement d'une séquence d'ARN

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, 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 5.5. *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 :

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

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 fnt sur les chaînes de la séquence des mots triés. Il y a donc $O(n^2)$ sous-problèmes pour une instance avec n mots. Une version mémoïsée de l'algorithme est donné en Figure 73, 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 exécutés 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 5.1. En reprenant l'exemple du début de section, nous obtenons les tableaux de la Figure 74.

```
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) + Σ_{k=i}^{j} p(k)
14        arbre_table[i][j] ← joint(arbre_table[i][r − 1], mots[r], arbre_table[r + 1][i])
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 73 – Calcul d'un arbre binaire de recherche optimal pour n mots triés. La probabilité de chaque mot est donné par le tableau p.

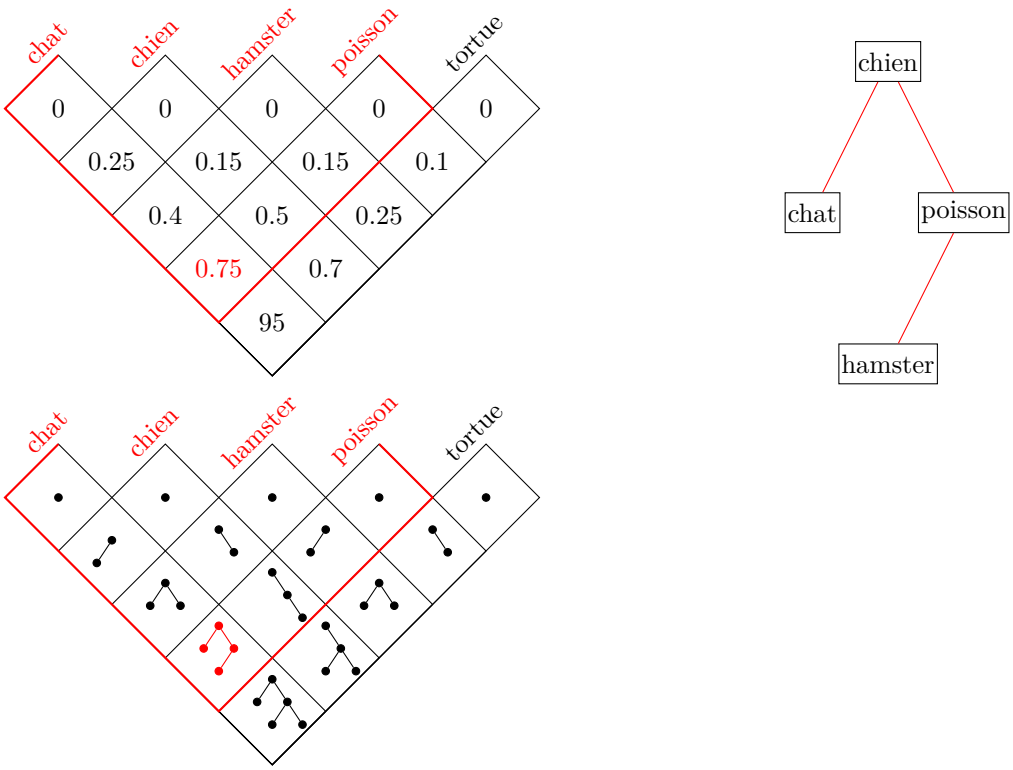


FIGURE 74 – Calcul de l'arbre optimum sur un exemple. Une des tables donne le coût optimal, l'autre donne l'arbre optimal (ici, seul la forme de l'arbre est donnée, l'arbre se déduit ensuite grâce à l'invariant des arbres binaires de recherche). Une case correspond à l'ensemble de mots de la pyramide 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.

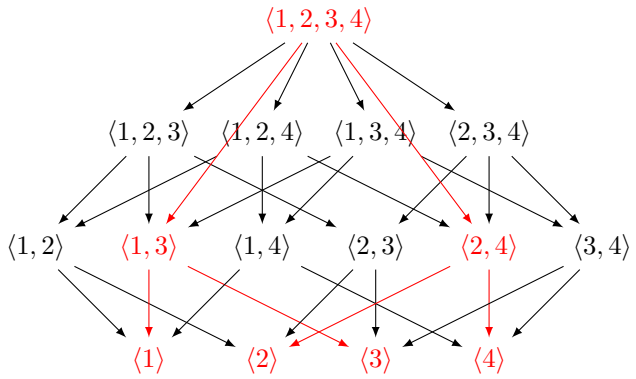


FIGURE 75 – Une toute petite partie de G_f pour le tri par fusion, et les sommets accessibles depuis $\langle 1, 2, 3, 4\rangle$ (tous les arcs ne sont pas dessinés).

5.3 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 $L =< e_0, \ldots, e_n >$, si $n \leq 0$, L est la liste triée, sinon nous trions les listes $L_0 =< e_0, e_2, \ldots >$ et $L_1 =< e_1, e_3, \ldots >$ 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 le cas de la liste $\langle 1, 2, 3, 4\rangle$ est donné en Figure 75.

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 5.1. *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 $T(n) = \Theta(f(n))$*

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 $O(n \log n)$.

5.3.1 Sélection du k^{e} élément

Soit $\mathsf{liste} = \langle e_1, \ldots, e_n \rangle$ 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 $|\{e_j : j \in [1, n], e_j <= e_i\}| = k$, 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 $|\{e_j : j \in [1, n], e_j <= e\}|$. Nous cherchons donc l'élément de rang k.

Une solution simple consiste à trier la séquence en un 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 $\mathsf{list\_inf}$, et celle des éléments supérieurs notée $\mathsf{liste\_sup}$.

Si $|\mathsf{liste\_inf}| > k$, le k^{e} élément de liste est aussi le k^{e} élément de $\mathsf{liste\_inf}$. Sinon, $\mathsf{liste\_sup}$ 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 $k - |\mathsf{liste\_inf}|^{\text{e}}$ élément de $\mathsf{liste\_sup}$. La Figure 76 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 $\mathsf{partition}$ 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

```
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 76 – Sélection du k^{e} élément.

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 suivant. 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 77, et une illustration est donné en Figure 78.

Nous analysons maintenant cet algorithme.

Lemme 5.6. *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. □

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

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

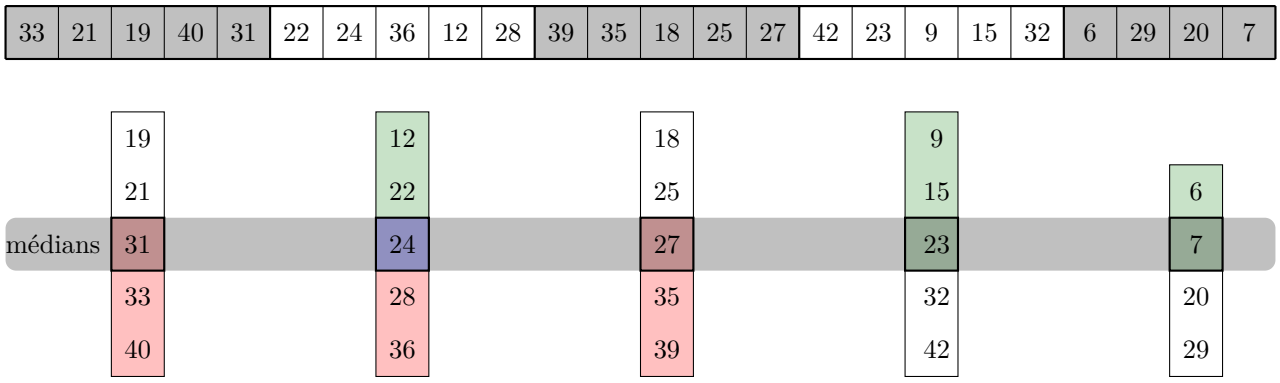


FIGURE 78 – Un exemple de sélection du pivot. La liste est coupée en morceaux de 5 (représentés ici verticalement) et chaque morceau de 5 est trié. Le médian des médians de chaque morceau est 24. La moitié des morceaux de 5 possède la moitié de leurs éléments supérieurs au pivot (en rouge), l'autre moitié possède une moitié de leurs éléments inférieurs au pivot (en vert).

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 $P(n) = O(n)$.

5.3.2 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 la plus proche ? 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 $n/2$ points de plus petites abscisses pour définir le premier sous-problème (gauche), et les $n - n/2 \sim n/2$ points d'abscisses les plus grandes pour

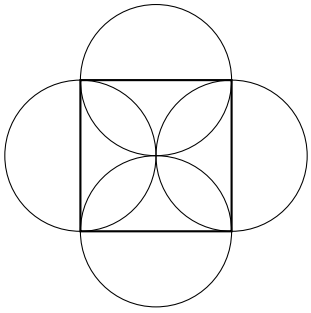


FIGURE 79 – Couverture d'un carré de côté 1 par 4 disques de diamètre 1

le second sous-problème (droit). Pour 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 5.7. *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 79), d'où le résultat. □

Lemme 5.8. *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 5.7, l'ensemble contient bien au plus 8 points. □

Ce Lemme 5.8 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 $O(n \log n)$.

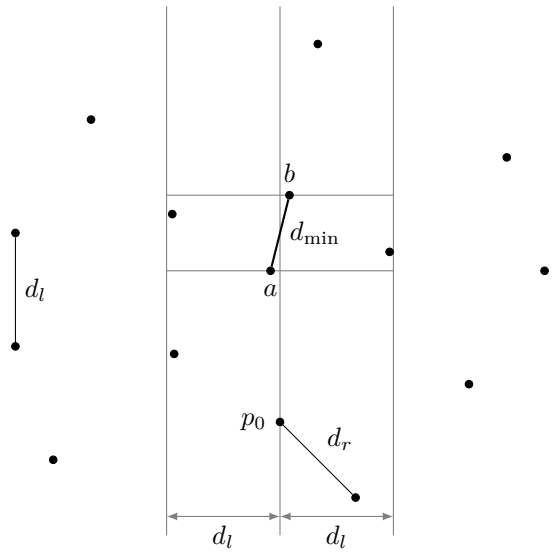


FIGURE 80 – Exemple de l'algorithme de recherche de la paire de points la plus proche. p_0 est le sommet médian en terme d'abscisse. d_l et d_r sont les plus courtes distances respectivement sur les points à gauche et à droite de p_0. $d_{\min}$ est la plus petite distance dans l'absolu. La bande verticale 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).

6 Analyse Amortie

Nous avons jusque-là principalement utilisé l'analyse asymptotique dans le pire des cas pour mesurer l'efficacité de nos algorithmes. C'est un choix qui est justifié par l'analyse des performances des algorithmes en pratique sur des exemples concrets, qui correspond le plus souvent à ce que prédit la théorie. Cependant, ce n'est pas une règle infaillible : certains algorithmes peuvent être bien plus rapides que ce que prédit une analyse dans le pire des cas, ne serait-ce que parce que le pire des cas n'est pas toujours représentatif des autres cas.

Les sujets de l'analyse amortie et de l'algorithmique randomisée est d'introduire une base théorique permettant d'obtenir des algorithmes plus performants que la prédiction du pire des cas. L'analyse amortie s'applique sur des problèmes pour lesquels les cas dans lesquels l'algorithme prend beaucoup de temps arrive suffisamment rarement pour ne pas avoir d'impact sur la complexité générale. L'algorithmique randomisé concerne des algorithmes utilisant du non-déterminisme via le choix de valeurs aléatoires, de sorte qu'avec très forte probabilité, le nombre d'opérations élémentaires est relativement faible. Il s'agit donc de deux techniques distinctes qui permettent de dépasser le point de vue trop limité de l'analyse du pire des cas. Nous commençons par étudier l'analyse amortie.

6.1 Un exemple : les files FIFO

Il existe de nombreuses structures concrêtes permettant d'implémenter la structure abstraite de file (*First In, First Out*), en particulier les listes simplement chainées ou l'utilisation d'un tableau avec deux indices flottants indiquant le début et la fin de la file. Ici, nous allons en dériver une depuis la structure abstraite de liste. Cela est particulièrement avantageux si nous disposons déjà de listes, ce qui est le cas dans certains langages, mais aussi nous obtiendrons une structure persistente : ajouter un élément à la file F crée une nouvelle file F', mais F est toujours utilisable et n'a pas changé.

Une file est une structure linéaire, qui encode donc une séquence. Les opérations principales sont :

— enfile(*file de t* f, t elt) : *file de t* qui insère un élément en fin de file,
— tête(*file de t* f) : t qui retourne le premier élément de la file,
— défile(*file de t* f) : *file de t* qui retire le premier élément de la file.

Formellement, en notant $[\![\mathsf{f}]\!] = \langle e_1, \dots, e_n \rangle$:

— $[\![\text{enfile}(\mathsf{f}, \mathsf{elt})]\!] = \langle e_1, \dots, e_n, [\![\mathsf{elt}]\!] \rangle$,
— $[\![\text{tête}(\mathsf{f})]\!] = e_1$,
— $[\![\text{défile}(\mathsf{f})]\!] = \langle e_2, \dots, e_n \rangle$.

Il conviendrait d'y ajouter la valeur file_vide : *file de t*, et le test est_vide(*file de t*) : *booléen*.

Tout comme les listes il s'agit donc d'une structure linéaire et persistente, la différence étant que l'insertion se fait en tête d'une liste, mais en fin d'une file. Nous pourrions tenter de coder une file en utilisant directement une liste, mais dans ce cas, l'insertion devient une opération complexe que nous pourrions coder ainsi :

```
1  fonction enfile(liste de t f, t elt) : liste de t =
2    si est_vide(f) alors retourner insère(elt, liste_vide)
3    sinon retourner insère(tête(f), enfile(queue(f), elt))
```

La fonction enfile est récursive et son premier argument voit sa longueur diminuée de 1 par l'aplel récursif, la récursion a donc une profondeur donnée par la longueur de la liste. La complexité asymptotique d'enfile sur une file de longueur n est donc $O(n)$. Ce n'est donc pas très efficace.

Pour coder l'insertion plus efficacement, nous allons utiliser une deuxième liste. Simplement, la file va être coupée en deux partie : un préfixe et le suffixe complémentaire. Le préfixe sera encodé tel quel comme une

liste préfixe, alors que le suffixe sera encodé en sens inverse dans une autre liste rev_suffixe. Ainsi, l'insertion dans la file se fait par insertion en tête de rev_suffix, ce qui prend un temps constant. Retirer l'élément de tête revient à supprimer le premier élément de préfixe.

Le problème est : que se passe-t-il si on souhaite défiler mais que préfixe est vide ? Dans ce cas, il faut retirer le dernier élément de rev_suffixe, ce qui est une opération coûteuse. Ce que nous allons faire dans ce cas, c'est reverser tous les éléments du suffixe vers le préfixe : le suffixe va ainsi devenir vide. L'algorithme est décrit en Figure 81. Nous utilisons un invariant supplémentaire : si la file est non-vide, alors son préfixe est non-vide également. La fonction pousse_à_gauche permet de vérifier l'invariant, sinon elle renverse le suffixe vers le préfixe. Cet invariant permet de ne pas avoir à faire de tests dans la fonction tête et défile.

Analysons cette structure. La fonction renverse est la seule non-triviale : c'est une fonction récursive tels que l'argument de l'appel récursif possède un suffixe de longueur un de moins que la file actuelle. Sa complexité est donc $O(s)$ où s est la longueur du suffixe.

Comme le suffixe peut dans le pire des cas contenir tous les éléments sauf un, nous en déduisons que les fonctions enfile et défile ont une complexité $O(n)$ où n est la nombre d'éléments total dans la file. Ce n'est donc pas mieux dans le pire des cas que d'utiliser juste une seule liste.

À l'utilisation, lorsque la structure de liste utilisée est efficace, ces files le sont aussi, c'est même la meilleure implémentation dans certains langages proposant des listes bien optimisées. Comment expliquer cela alors que notre analyse donne une complexité linéaire aux opérations enfile et défile ? Tout simplement : c'est l'opération renverse qui coûte cher, mais elle est exécutée très rarement : chaque élément inséré ne sera renversé qu'une seule fois.

Ainsi, même si une opération individuelle sur ces files peut être coûteuse, pour arriver à un tel coût il a fallu faire de nombreux calculs avant, ici suffisamment d'insertions, ce qui *amortit* le prix de l'opération coûteuse. L'idée de l'analyse amortie est donc de tenir compte de toutes les opérations précédentes : plutôt que de regarder le coût d'une opération unique, nous allons calculé le coût d'une séquence d'opérations.

Définition 6.1. *Soit E un ensemble d'états, et $\phi : E \to \mathbb{R}^+$ une* fonction de potentiel. *Le coût amorti d'une opération $e \to e'$ sous le potentiel ϕ est défini par*

$$\textit{coût amorti}(e \to e') := \textit{coût réel}(e \to e') + \phi(e') - \phi(e)$$

Le coût amorti sour le potentiel ϕ d'une séquence d'opérations $e_0 \to e_1 \ldots \to e_n$ est la somme des coûts amortis de chaque opération.

Dans notre exemple, les états sont les files concrêtes possibles, donc des paires de listes, et le coût réel est le coût algorithmique, en nombre d'opérations élémentaires, des opérations sur les files. La fonction de potentiel reste à trouver, l'idée étant que plus une opération a un coût réel élevé, plus il faudra qu'elle fasse diminuer le potentiel de l'état. Nous pouvons donc déjà imaginer que le potentiel doit dépendre de la longueur du suffixe.

Le potentiel représente de l'épargne. Le principe est d'épargner à chaque opération peu coûteuse. Ainsi, lorsque nous devons faire une opération plus chère, nous pouvons puiser dans l'épargne pour la financer.

Proposition 6.1. *Le coût amorti sous le potentiel ϕ d'une séquence d'opérations $e_0 \to e_1 \ldots \to e_n$ est égal au coût réel de la séquence plus $\phi(e_n) - \phi(e_0)$.*

Démonstration. On applique la définition et on élimine les termes opposés dans la somme. □

Une conséquence est que si $\phi(e_0) = 0$, le coût amorti est une borne supérieur du coût réel de la séquence d'opérations. Ce qui est exactement ce qu'il nous faut pour comprendre les performances réelles des algorithmes.

```
1   type file de t = {
2       préfixe : liste de t;
3       rev_suffixe : liste de t
4     }
5
6   fonction renverse(file de t file) : file de t =
7     si Liste.est_vide(file.rev_suffixe) alors retourner file
8     sinon retourner{
9       préfixe = Liste.insère(Liste.tête(file.rev_suffixe), file.préfixe);
10      rev_suffixe = Liste.queue(file.rev_suffixe)
11    }
12
13  fonction pousse_à_gauche(file de t file) : file de t =
14    si Liste.est_vide(file.préfixe)alors retourner renverse(file)
15    sinon retourner file
16
17  fonction file_vide =
18    retourner{
19      préfixe = liste_vide;
20      rev_suffixe = liste_vide
21    }
22
23  fonction est_vide(file de t file) : booléen =
24    retourner Liste.est_vide(file.préfixe) ∧ Liste.est_vide(file.rev_suffixe)
25
26  fonction enfile(file de t file, t elt) : file de t =
27    retourner pousse_à_gauche({
28      préfixe = file.préfixe;
29      rev_suffixe = Liste.insère(elt, file.rev_suffixe)
30    })
31
32  fonction défile(file de t file) : file de t =
33    retourner pousse_à_gauche({
34      préfixe = Liste.queue(file.préfixe);
35      rev_suffixe = file.rev_suffixe
36    })
37
38  fonction tête(file de t file) : t =
39    retournerListe.tête(file.préfixe);
```

FIGURE 81 – Implémentation d'une file par deux listes.

Reprenons la file. Il faut faire payer au potentiel (notre épargne) le coût des renversements. Notons cs une borne supérieure de la complexité d'un renversement d'une file ayant un suffixe de longueur s. Cette complexité dépend linéairement de la longueur du suffixe à renverser. Le potentiel doit donc être aussi linéaire en cette longueur. Pour une file file avec un préfixe de longueur p et un suffixe de longueur s, nous posons donc :

$$\phi(\mathsf{file}) = cs$$

Notons que le potentiel de la file vide est 0.

Nous calculons maintenant le coût amorti de chaque opération non-triviale :

— $\mathsf{enfile} : f \to f'$ (sans renversement) fait augmenter la longueur du suffixe de 1, son coût amorti est donc $O(1) + \phi(f') - \phi(f) = O(1) + c = O(1)$,

— $\mathsf{défile} : f \to f'$ (sans renversement) ne change pas la longueur du suffixe. Son coût amorti est donc $O(1) + \phi(f') - \phi(f) = O(1)$,

— $\mathsf{enfile} : f \to f'$ (avec renversement) ne se produit que si f est la file vide. Le coût amorti est alors $O(1) + \phi(f') - \phi(f) = O(1) + c = O(1)$.

— $\mathsf{défile} : f \to f'$ (avec renversement) part d'un suffixe de s éléments et produit une file dont le suffixe est vide. Nous avons donc que le coût amorti est $O(1) + cs + \phi(f') - \phi(f) = O(1) + cs + 0 - cs = O(1)$.

Ainsi toutes les opérations ont un coût amorti constant ! Puisque la file vide a un potentiel de 0, le coût amorti de n'importe quelle séquence d'opérations en partant de la liste vide est donc linéaire en le nombre de ces opérations, tout comme c'est le cas pour les files encodées par les listes simplement chainées ou des tableaux. Ceci explique donc pourquoi cette implémentation est compétitive avec les implémentations ayant des complexités constantes dans le pire des cas.

6.2 Tables dynamiques

Comment construire une table de dispersion sans connaitre *a priori* le nombre d'éléments qu'elle contiendra ? Il faut respecter le principe de ne pas utiliser plus d'espace que nécessaire (à une constante multiplicative près), tout en évitant d'avoir trop de collision, donc d'utiliser un tableau de trop petite taille. Cela oblige à régulièrement redimensionner le tableau et donc à réindexer toutes les clés : ce qui est une opération fort coûteuse : linéaire en le nombre de clés.

Pourtant, les tables de dispersion, nous l'avons vu, sont des structures redoutablement efficaces. Leur secret vient aussi de l'analyse amortie !

Rappelons que le paramêtre $\alpha = \frac{k}{n}$ est le taux de remplissage de la table, avec k le nombre de clés stockées, et n la longueur du tableau. Nous voulons imposer que α soit toujours dans un intervalle étroit autour de 1. Pour cela, nous allons utiliser la règle suivante :

— si suite à une insertion, $\alpha > 2$, doubler la taille du tableau et réindexer les éléments.

— si suite à une suppression, $\alpha < 1/2$, diviser par deux la taille du tableau.

Dans les deux cas, après à un redimensionnement, α vaut approximativement 1.

Les opérations coûteuses sont donc les redimensionnements, nous supposerons que l'insertion ou la suppression d'un élément a une complexité dans le pire des cas de c opérations élémentaires (c'est une simplification, puisque dans une table de dispersion, le coût est constant en espérance seulement). Un redimensionnement a donc comme complexité $cn + dn$ où le terme cn correspond au coût d'initialisation d'un nouveau tableau de taille n. Notons $b = c + d$.

Les redimensionnements arrivent lorsque le nombre d'éléments diminue trop, où au contraire lorsqu'il augmente trop. Au contraire, lorsque α est proche de 1, nous savons qu'il est possible de faire un grand nombre d'opérations avec le prochain redimensionnement.

Le choix de la fonction de potentiel doit refléter cette discussion : si α est proche de 1, le potentiel doit être faible, si α se rapproche de 1/2 ou 2 il doit être élevé. Nous utilisons donc une fonction linéaire par morceau :

$$\phi(k,n) = \begin{cases} 2b(k-n) & \text{si } k \geq n \\ b(n-k) & \text{si } k < n \end{cases}$$

Nous pouvons maintenant analyser le coût amorti de chaque opération, partant d'une table de taille n contenant k clés :

— insertion sans redimensionnement $t \to t'$ et $k \geq n$: $c + \phi(t') - \phi(t) = c + 2b = O(1)$
— insertion sans redimensionnement $t \to t'$ et $k < n$: $c + \phi(t') - \phi(t) = c - b = O(1)$
— suppression sans redimensionnement $t \to t'$ et $k \geq n$: $c + \phi(t') - \phi(t) = c - 2b = O(1)$
— suppression sans redimensionnement $t \to t'$ et $k < n$: $c + \phi(t') - \phi(t) = c + b = O(1)$
— insertion avec redimensionnement $t \to t'$: $2nd + c(2n+1) + \phi(t') - \phi(t) \leq 2bn + c + 0 - 2bn = O(1)$
— suppression avec redimensionnement $t \to t'$: $dn/2 + cn/2 + \phi(t') - \phi(t) \leq bn/2 + 0 - bn/2 = O(1)$

Le coût amorti de chaque opération est donc constant ! Les coûts de redimensionnement sont donc amortis par les insertions et suppressions nécessaires entre deux redimensionnements.

6.3 Incrément binaire

Un exemple classique d'analyse amortie est l'incrément des entiers codés en binaire par une liste de bits 0 ou 1. En effet, l'incrément revient à trouver le plus long suffixe de la forme $011\ldots1 = 01^k$ et de le remplacer par $100\ldots0 = 10^k$. Dans le pire des cas, cette opération prend un temps linéaire en le nombre de bits, si par exemple l'entier est $2^n - 1$.

Nous pouvons montrer que le temps amorti de l'incrément est en fait constant pour le potentiel $\phi(n) = c \times$nombre de 1 dans l'écriture binaire de n. Avec les notations ci-dessus, le nombre d'opérations élémentaires effectuées est $c(k+1)$, et la différence de potentiel entre les deux configurations est ck, pour un coût amorti de $c = O(1)$.

L'application de cette analyse concerne les tas binomiaux, une structure concrête de file de priorité basée sur le système de numérotation binaire. La complexité de l'insertion dans un tas binomial est exactement donnée par celle de l'incrément binaire, donc l'insertion dans un tas binomial a une complexité amortie constante (pour le choix de la fonction de potentiel appropriée).

7 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 étudier quelques exemples. Mais avant cela, comme beaucoup d'algorithmes probabilistes commencent par une étape de mélange des données d'entrée, nous commençons par rappeler comment mélanger un tableau de valeur en temps optimal.

7.0.1 Mélanger un tableau uniformément

Étant donné un tableau $\mathtt{t}[1..n]$ contenant des valeurs d'un type quelconque, comment mélanger ces valeurs de façon uniforme, c'est-à-dire telle que tout permutation des valeurs possède la même probabilité d'être tirée ?

On peut définir l'ensembe des permutations d'une séquence $\langle t_1, t_2, \ldots, t_n\rangle$ par

$$\begin{aligned}\text{permutations}(\langle t_1, t_2, \ldots, t_n\rangle) &= \bigcup_{i=1}^{n} t_i \cdot \text{permutations}(\langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\rangle)\\ \text{permutations}(\langle\rangle) &= \{\langle\rangle\}\end{aligned}$$

Nous en déduisons immédiatement :

$$\begin{aligned}|\text{permutations}(\langle t_1, t_2, \ldots, t_n\rangle)| &= |\bigcup_{i=1}^{n} t_i \cdot \text{permutations}(\langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\rangle)|\\ &= \sum_{i=1}^{n} |\text{permutations}(\langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\rangle)|\\ &= n \times |\text{permutations}(\langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\rangle)|\\ &= n! \qquad (\text{car } |\text{permutations}(\langle\rangle)| = 1)\end{aligned}$$

De plus tout élément a donc la même probabilité d'être le premier élément d'une permutation choisie aléatoirement (la probabilité est $1/n$).

Nous pouvons aussi facilement voir que l'ordre des éléments dans l'argument de permutations n'a pas

```
1    // Version récursive dérivée de la formule mathématique
2  fonction permuter(tableau[deb..fin] de t tableau) : void =
3    si deb = fin alors retourner
4    soit i := entier aléatoire entre deb et fin
5    tableau[deb] ↔ tableau[i];
6    permuter(tableau[deb + 1..fin]);
7
8    // Version impérative dérivée de la version récursive
9  fonction permuter(tableau[1..fin] de t tableau) : void =
10   pour deb de 1 à fin − 1 faire
11     soit i := entier aléatoire entre deb et fin
12     tableau[deb] ↔ tableau[fin]
```

FIGURE 82 – Mélanger un tableau aléatoirement

d'importance :

$$\begin{aligned}
\text{permutations}(\langle t_1, t_2, \ldots, t_n\rangle) &= \bigcup_{i=1}^{n} t_i \cdot \text{permutations}(\langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\rangle) \\
&= \bigcup_{i=1}^{n} \bigcup_{j=1, j\neq i}^{n} t_i \cdot t_j \cdot \text{permutations}(\langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_{j-1}, t_{j+1}, \ldots, t_n\rangle) \\
&= \bigcup_{j=1}^{n} \bigcup_{i=1, i\neq j}^{n} t_i \cdot t_j \cdot \text{permutations}(\langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_{j-1}, t_{j+1}, \ldots, t_n\rangle) \\
&= \bigcup_{j=1}^{n} t_i \cdot \text{permutations}(\langle t_1, \ldots, t_{i-1}, t_j, t_{i+1}, \ldots, t_{j-1}, t_{j+1}, \ldots, t_n\rangle) \\
&= \text{permutations}(\langle t_1, \ldots, t_{i-1}, t_j, t_{i+1}, \ldots, t_{j-1}, t_i, t_{j+1}, \ldots, t_n\rangle)
\end{aligned}$$

Nous pouvons donc réécrire :

$$\begin{aligned}
\text{permutations}(\langle t_1, t_2, \ldots, t_n\rangle) &= \bigcup_{i=1}^{n} t_i \cdot \text{permutations}(\langle t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n\rangle) \\
&= \bigcup_{i=1}^{n} t_i \cdot \text{permutations}(\langle t_2, \ldots, t_{i-1}, t_1, t_{i+1}, \ldots, t_n\rangle)
\end{aligned}$$

Et nous en déduisons l'algorithme de la Figure 82. Nous analysons cet algorithme que nous avons d'abord décrit sous la forme récursive suggérée par la formule précédente. Comme la valeur de deb augmente de 1 à chaque étape, partant de 1 jusqu'à valoir fin, nous pouvons simplifier l'algorithme en faisant directement appel à une boucle **for**. La complexité s'en déduit immédiatement : $\Theta(n)$ opérations élémentaires pour un tableau de longueur n.

Notons que le nombre de bits générés aléatoirement qu'il faut pour mélanger une permutation est de $\Omega(\log n!) = \Omega(n \log n)$, puisqu'il faut choisir entre $n!$ permutations et chaque bit permet de couper par deux

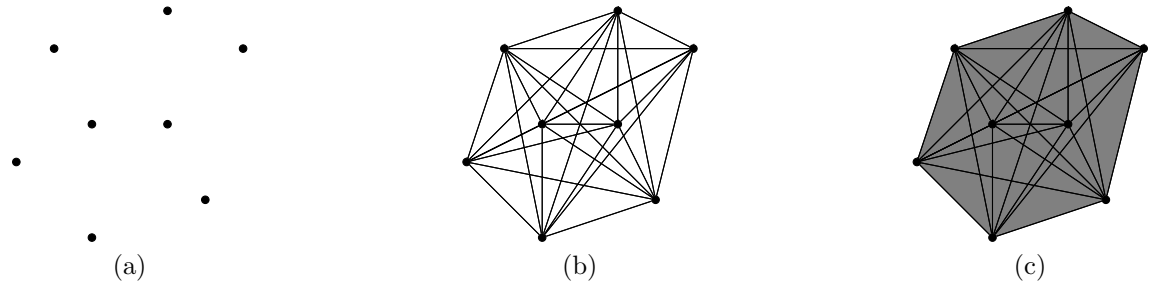


FIGURE 83 – L'enveloppe convexe d'un ensemble fini de points

le nombre de permutations encore possibles. Dans notre algorithme, chaque entier généré prend 32 ou 64 bits, ce qui est supérieur à $\log n$ pour les valeurs de n qui nous intéressent.

7.0.2 Enveloppe convexe de points du plan

Définition 7.1. *Un ensemble R de points du plan Euclidien est* convexe *si pour tous points $a, b \in R$, le segment $[a, b]$ est contenu dans R.*

*Soit S un ensemble de points dans le plan Euclidien. L'*enveloppe convexe *conv(S) de S est le plus petit ensemble convexe contenant S.*

Étant donné un ensemble fini S de points, une façon simple de construire mentalement l'enveloppe convexe de S est de tracer tous les segments définis par ces points, et de remplir les régions bornées créées par ces segments, comme le montre la Figure 83.

Nous nous demandons comment produire cette enveloppe convexe. Nous souhaitons produire la liste des sommets au bord de l'enveloppe dans l'ordre trigonométrique. Nous pourrions pour cela garder la liste des segments qui se trouvent sur le bord et les trier d'une certaine façon. Cela nécessite de produire tous les segments possibles, et l'algorithme obtenu aurait une complexité de $O(n^2)$ pour un ensemble de n points.

Notons $S := \{p_1, p_2, \ldots, p_n\}$ la liste des points du plan. Nous donnons maintenant un algorithme en $O(n \log n)$, dont le principe est de commencer par calculer l'enveloppe convexe des trois premiers points, puis des 4 premiers, et ainsi de suite en ajoutant un seul point à chaque étape. Ainsi, l'algorithme au cours de son exécution maintient l'enveloppe convexe des i premiers points, qui est donc une séquence de sommet d'un polygone convexe, nous appelons cette séquence polygone. Elle est encodé par une liste doublement chainée : nos disposons donc d'une fonction suivant et d'ue fonction précédant pour obtenir les éléments adjacents à un sommet.

Nous allons aussi utiliser un autre invariant. Nous allons garder un point intérieur au polygone, qui restera fixe pendant tout l'algorithme (nous commencerons avec un polygone triangulaire, nous pourrons donc prendre son centre de gravité par exemple), que nous appelons donc centre. En utilisant les côtés du polygone, nous pouvons tracer des cônes issus de centre, qui partitionnent les points $\{p_{i+1}, p_{i+1}, \ldots, p_n\}$. Chacun de ces points est donc associé à un des segments du polygone :

- si p_j est un sommet du polygone et $p_{j'}$ le sommet suivant dans le sens trigonométrique, cone$[j]$ est la liste des points parmi $p_{i+1}, \ldots, p_n$ contenus dans le cône associé au segment $[p_j, p_{j'}]$,
- si $j > i$, segment$[j]$ est l'indice de l'extrémité droite (vu de centre) du segment auquel est associé p_j (de sorte que cone[segment$[j]$] contienne j).

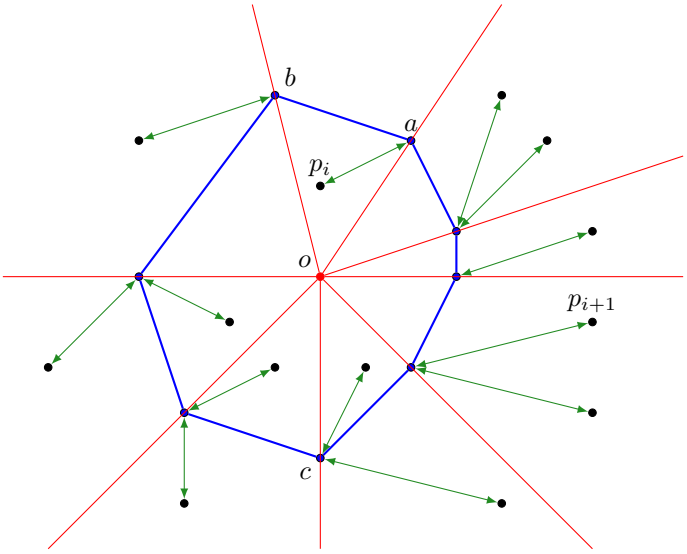


FIGURE 84 – Représentation des structures maintenues par l'algorithme, avant l'étape i. Le polygone bordant l'enveloppe convexe des points déjà traités est en bleu. Le point rouge o est un point fixe à l'intérieur de l'enveloppe convexe : il génère des cônes pour chaque segment du polygone bleu. Les points non-traités sont chacun associés à exactement un point du polygone.

La Figure 84 montre la structure ainsi entretenue par l'algorithme.

Maintenant que nous avons décrit les structures maintenus par l'algorithme lors de son exécution, nous devons déterminer comment traiter un point pas encore considéré. Lors de l'itération i, l'algorithme doit prendre en compte le point p_i. Deux cas sont possibles :

— le point p_i est déjà à l'intérieur de polygone, qui délimite l'enveloppe convexe de $p_1, \ldots, p_{i-1}$. Dans ce cas, l'enveloppe convexe pour les points p_1 à p_i est aussi délimitée par polygone. De plus, il est facile de détecter ce cas, puisqu'il suffit de déterminer si p_i est à l'intérieur du triangle de sommets centre, segment[i], suivant(segment[i]). C'est le cas du point nommé p_i de la Figure 84, qui appartient à l'intérieur du triangle oab. Nos obtenons alors la Figure 85. La complexité pour le traitement de ce cas est $O(1)$.

— Sinon le point p_i est à l'extérieur de ce triangle. Dans ce cas, il nous faut calculer un polygone plus grand, passant par p_i. Nous cherchons pour cela tous les segments de polygone tel que p_i soit du côté extérieur de la droite prolongeant ces segments. Ce sont des segments consécutifs du polygone, entre deux sommet q_1 et q_2. Nous les supprimons de polygone et les remplaçons par les segments $[q_1, p_i]$ et $[q_2, p_i]$. Cela oblige de recalculer toutes les associations entre points non traités et cônes pour les segments supprimés, ce qui ce fait en temps constant par points à replacer. Ce cas est illustré par le point p_{i+1} de la Figure 85, donnant la Figure 86

Nous devons analyser la complexité du deuxième cas. Nous pouvons subdiviser les calculs ainsi pour analyser indépendamment les complexités de chaque point :

(1) Trouver q_1 et q_2, et mettre à jour la liste des segments de polygone.

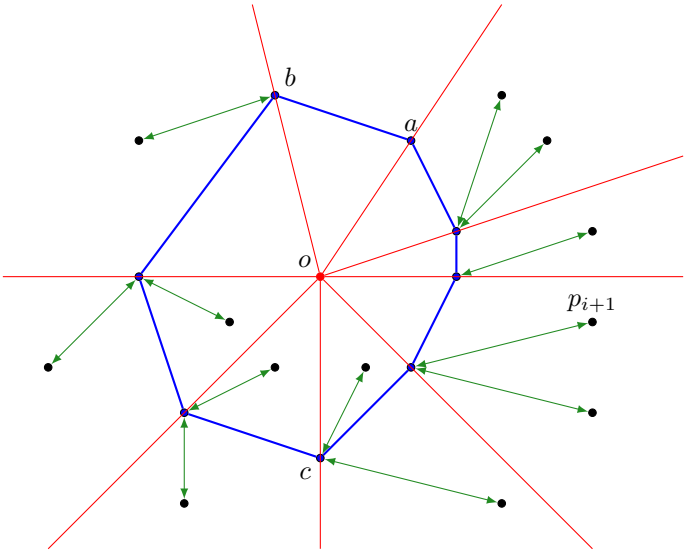


FIGURE 85 – Juste après l'étape i. p_i étant déjà dans le polygone, celui-ci ne change pas. Nous supprimons donc simplement p_i.

(2) Recalculer les associations pour les points se trouvant dans les cônes supprimés.

Le point (1) se fait par une analyse amortie, en prenant pour $\phi(\mathsf{polygone})$ une valeur proportionnelle au nombre de segments constituant le polygone. Le nombre de segments s parcourus pour trouver q_1 et q_2, qui sont aussi les segments supprimés, caractérise la complexité de cette mise-à-jour, prenant en complexité réelle $O(s)$. Le potentiel ϕ pendant l'opération est modifié de $O(2-s)$. Avec le choix de la même constante pour le potentiel, on obtient une complexité amortie de $O(s+2-s) = O(1)$.

Pour le point 2, il nous faut calculer le nombre de points dont le segment est recalculé. Notons S_i l'ensemble des i points traités avant et pendant l'itération ajoutant p_i. Puisque les points sont traités dans un ordre uniformément aléatoire, étant fixé S_3 le triangle initial, la probabilité $\Pr[p_i = q]$ pour tout point $q \in S_i \setminus S_3$ est uniformément égale à $1/(i-3)$. Pour un point t non-contenu dans S_i, la probabilité que son segment soit modifié à l'itération i est égale à la probabilité qu'une des extrémités de son segment associé dans $\mathrm{conv}(S_i)$ soit p_i. Cette probabilité est donc de $2/(i-3)$. L'espérance du coût pour chaque point non-traité à l'itération est donc $O(1/i)$.

En sommant sur tous les points et toutes les itérations (par linéarité de l'espérance), on obtient que le nombre espéré de mise-à-jours des associations points-segments est :

$$\sum_{i=4}^{n}(n-i)O\left(\frac{1}{i}\right) \leq O\left(n\sum_{i=1}^{n}\frac{1}{i}\right) = O(n\log n)$$

Théorème 7.1. *Le coût asymptotique espéré de cet algorithme de calcul d'une enveloppe convexe de n points est $O(n\log n)$.*

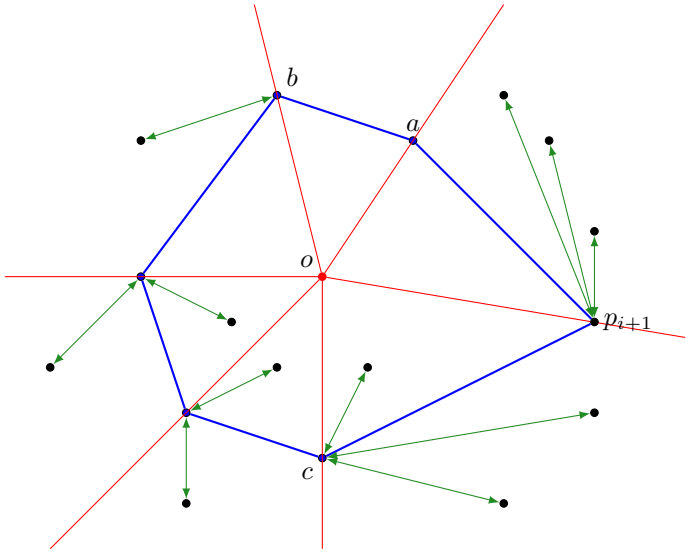


FIGURE 86 – Après le traitement de p_{i+1}. Tous les segments entre c et a doivent être supprimés et remplacés par $[c, p_{i+1}]$ et $[p_{i+1}, a]$.

7.0.3 Coupe minimum dans un graphe

Soit $G = (V, E)$ un graphe (non-orienté) et $c : E \to \mathbb{N}+$ une fonction de capacité. Nous avons vu lors du calcul du flot maximum qu'il est possible de calculer la st-coupe (orienté) d'un graphe (orienté) de capacité minimum d'un graphe. Nous pouvons utiliser cet algorithme pour calculer une plus petite coupe : un ensemble de sommets $U \subsetneq V$, $U \neq \emptyset$, qui minimise $\sum_{e \in \delta(U)} c_e$. Ici $\delta(U) = \{e = uv \mid u \in U, v \notin U\}$. Pour cela, il suffit de remplacer chaque arête non-orienté e par deux arcs entre la même paire de sommets, l'un dans chaque sens, ayant la même capacité que $c(e)$. Ensuite, il faut tester toutes les paires st possibles et garder celes 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 utilisé une source d'aléa pour calculer 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. Précisément :

Lemme 7.1. *Notons $C := \sum_{e \in E} c_e$. Soit k la capacité minimum d'une coupe de G. Alors $C \geq kn/2$.*

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 :

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

Par ailleurs, dans cette sommation, chaque arête est comptée deux fois, une fois à chacune de ses extrêmités.

Ce qui donne :

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

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

Ainsi :

Lemme 7.2. *Soit U une coupe minimum de G. Soit e un arête choisi aléatoirement avec une distribution proportionnelle aux capacités des arcs (c'est-à-dire $\Pr[e \text{ est choisie}] = c_e/C$). Alors la probabilité que e soit dans la coupe minimum considérée est*

$$\Pr[e \in \delta(U)] \leq 2/n.$$

Démonstration.

$$\Pr[e \in \delta(u)] = \sum_{e \in \delta(u)} \frac{c_e}{c} = \frac{k}{C} \leq \frac{2}{n}.$$

□

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 possible toutes les arêtes sont dans au moins une coupe minimum. Il est donc important ici que U soient une coupe minimum fixée, et ce que nous allons montrer, 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 extrêmités en un seul sommet.

Définition 7.2. *Soit $G = (V, E)$ un multi-graphe et $e = uv \in E$. Le graphe G/e,* contraction *du graphe G par l'arête e est le multi-graphe (V', E') avec :*

- *$V' := V \setminus \{v\}$,*
- *$E' := E \setminus \{e\}$,*
- *$dst_{G'}(e) = dst_G(e)$ 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 uv disparait, les autres arcs incidents à v sont incidents à u dans G/uv. Notons que toute coupe $\delta(U')$ dans $G \setminus e$ 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 choisissent 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 :

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

En répétant l'algorithme $O(n^2 \log n)$ fois, une des coupes de plus petite capacité trouvées est très probablement U.

8 Recherche de motifs

La recherche de motifs est une opération essentielle en informatique. Elle apparait dans tous les éditeurs de textes sérieux, ou comme outil à part entière, par exemple `grep` et ces dérivés sous Unix. Il s'agit de chercher un motif, qui peut être un mot, un ensemble de mots ou une expression régulière, dans un texte potentiellement extrêmement long (millions ou même milliards de caractères).

8.1 Recherche d'un mot dans un texte

Soit w un mot de longueur m, le motif à chercher, et t un texte de longueur n, dans lequel w est cherché. En général, w est donc un tableau indicé de 1 à m (ou 0 à $m-1$), et le texte est aussi un tableau, indicé de 1 à n. La question de la recherche du motif w dans le texte t revient à déterminer :

$$\{i \in [\![1, n-m+1]\!] \mid w[1..m] = t[i..i+m-1]\}$$

autrement dit, l'ensemble des occurences du motif w dans t, représentés par l'indice de leur première lettre dans t.

8.1.1 Algorithme naïf

L'algorithme naïf consiste à tester toutes les valeurs possibles pour $i \in [\![1, n-m+1]\!]$, et pour chacune faire tester l'égalité entre w et $t[i..i+m-1]$. Chaque test d'égalité prend un temps $O(m)$, et l'algorithme fait $n-m$ de ces tests, sa complexité est donc $O(m(n-m))$ dans le pire des cas.

Nous pouvons par exemple choisir $w = a^{m-1}b$ et $t = a^{n-1}b$ pour que ce pire des cas soit atteint :

a	a	a	a	a	a	a	a	a	a	a	a	a	a	a	b
a	a	a	a	!											
	a	a	a	a	!										
		a	a	a	a	!									
			a	a	a	a	!								
				a	a	a	a	!							
					a	a	a	a	!						
						a	a	a	a	!					
							a	a	a	a	!				
								a	a	a	a	!			
									a	a	a	a	!		
										a	a	a	a	!	
											a	a	a	a	b

8.1.2 Algorithme de Knuth-Morris-Pratt

L'algorithme naïf est trop dispendieux, car lorsqu'il échoue un test d'égalité, il n'utilise pas le résultat des tests entre deux caractères qui ont réussi, mais il oublie tout sons calcul.

L'algorithme de Knuth, Morris et Pratt vise à corriger ce défaut : si lors de l'itération $i \in [\![1, n-m+1]\!]$, l'algorithme trouve que les j premières lettres de w sont bien $t[i..i+j-1]$, mais que $w[j+1] \neq t[i+j]$, nous pouvons en déduire que le motif n'apparait pas en position i. Mais nous connaissons maintenant les j prochaines lettres. Si un motif pour w utilise certaines de ces lettres, disons k, il faut que le suffixe de

```
1   fonction prétraiteKMP(tableau[1..m] de caractères motif) : tableau[0..m] d'entiers =
2     soit L := tableau[0..m] d'entiers
3     soit longueurSuffixe := ref 0
4
5     soit la fonction défausseSuffixesInvalides(caractère prochaineLettre) : void =
6       tant que !longueurSuffixe ≥ 0 ∧ motif[!longueurSuffixe + 1] ≠ prochaineLettre faire
7         longueurSuffixe ← L[!longueurSuffixe]
8       retourner
9
10    L[0] ← −1; L[1] ← 0
11    pour i de 2 à m faire
12      défausseSuffixesInvalides(motif[i])
13      longueurSuffixe ← longueurSuffixe + 1
14      L[i] ← !longueurSuffixe
15    retourner L
```

FIGURE 87 – Prétraitement du motif dans l'algorithme de Knuth, Morris et Pratt

longueur k de $t[i..i+j-1]$ soit un préfixe de longueur k de w. Or $t[i..i+j-1] = w[1..j]$, cela revient donc à dire que $w[i+j-k..i+j-1] = w[1..k]$.

L'algorithme de Knuth-Morris-Pratt commence donc par calculer pour toute position $k \in [\![1, m]\!]$, la longueur du plus long suffixe propre de $w[1..k]$ qui soit préfixe de w (possiblement 0). Notons $L(k)$ cette longueur. Or, si $w[k'..k]$ est préfixe de w, alors assurément $w[k'..k-1]$ l'est aussi. Donc :

$$L(k+1) = \begin{cases} L(k)+1 & \text{si } w[k+1] = w[L(k)+1], \text{ sinon :} \\ L(L(k))+1 & \text{si } w[k+1] = w[L(L(k))+1], \text{ sinon :} \\ L(L(L(k)))+1 & \text{si } w[k+1] = w[L(L(L(k)))+1], \text{ sinon :} \\ \dots & \\ 0 & \text{sinon} \end{cases}$$

Le calcul de L est donc assuré par l'algorithme de prétraitement présenté en Figure 87. Dans l'algorithme, prochaineLettre prend la valeur de $w[k+1]$, et longueurSuffixe prend successivement les valeurs $L(k)$, $L(L(k))$, ..., selon les itérations de la boucle **tant que**. Ceci nécessite de calculer les valeurs de L de la plus petite à la plus grande, justifiant l'utilisation d'une boucle **pour**.

Une fois L calculé, il suffit de lire le texte un caractère à la fois (boucle sur i dans l'algorithme de la Figure 88), en retenant le plus long suffixe lu qui soit un préfixe du motif. Nous appelons sa longueur longueurSuffixe. La partie délicate est lorsque de la lecture d'un nouveau caractère, d'indice $i = [\![\text{i}]\!]$. Juste après l'augmentation de i (lignes 6 et 7), il faut mettre à jour la longueur d'un plus long suffixe valide. Si la lettre d'indice i correspond à la lettre suivante dans le motif, la longueur du suffixe augmente de 1. Sinon, il faut changer de suffixe : c'est là que L est utilisé : on applique L jusqu'à ce que la lettre d'indice i coïncide avec la lettre suivante du motif. Éventuellement, on peut trouver qu'aucun suffixe ne convient, en obtenant $L(1) = -1$. Puisque longueurSuffixe est ensuite incrémenté, dans ce cas on obtiendra un suffixe de longueur 0.

La complexité de ces deux algorithmes se calcule facilement par analyse amorti, en utilisant comme potentiel $[\![\text{longueurSuffixe}]\!]$. Ce potentiel est augmenté de 1 par chaque itération des boucles **for**, et diminue

```
1   fonction rechercheKMP(tableau[1..m] de caractères motif, tableau[1..n] de caractères texte) :
2           liste d'entiers =
3     soit trouvés := ref Liste.vide()
4     soit L := prétraitement(motif)
5     soit longueurSuffixe := ref 0
6
7     soit la fonction défausseSuffixesInvalides(caractère prochaineLettre) : void =
8       tant que !longueurSuffixe ≥ 0 ∧ motif[!longueurSuffixe + 1] ≠ prochaineLettre faire
9         longueurSuffixe ← L(!longueurSuffixe)
10      retourner
11
12    pour i de 1 à n faire
13      défausseSuffixesInvalides(motif[i])
14      longueurSuffixe ←!longueurSuffixe + 1
15      si !longueurSuffixe = m alors
16        trouvés ← Liste.insère(i − m + 1, !trouvés)
17        longueurSuffixe ← L(!longueurSuffixe)
18    retourner trouvés
```

FIGURE 88 – Recherche de motif par l'algorithme de Knuth, Morris et Pratt.

à chaque itération des boucles **tant que**. Chaque itération prend alors un temps amorti constant, et le potentiel est initialement nul et reste positif. La complexité du prétraitement est donc $O(m)$ et celle de la recherche $O(n)$.

Voici un exemple de prétraitement, pour le motif `ababcab` :

	0	1	2	3	4	5	6	7
motif		a	b	a	b	c	a	b
L	-1	0	0	1	2	0	1	2

et un exemple de recherche de ce motif. Les points représentent les lettres qui ne sont pas relues par l'algorithme, mais détectées grâce au calcul de L :

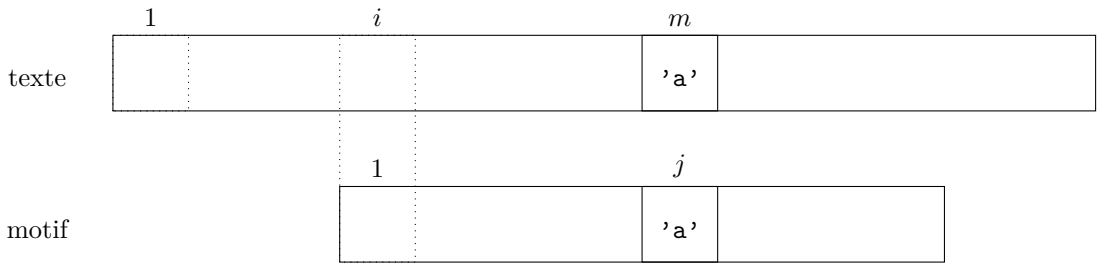
```
a a b a b a b c a b a a b a b c
a !
  a b a b !
        . . a b c a b                    motif trouvé !
                  . . a !
                      a b a b c
```

8.1.3 L'algorithme de Boyer-Moore

L'algorithme de Knuth, Morris et Pratt parvient à passer un temps constant sur chaque lettre, et il n'est pas possible de faire mieux en théorie. Cependant, certains algorithmes permettent de bien meilleurs résultats en pratique.

Le principe de l'algorithme de Boyer-Moore est de comparer la m^{e} lettre du motif et celle du texte : a-t-on $w[m] = t[m]$? Si l'égalité est vrai, nous testons la lettre précédente, etc. Si l'égalité est fausse, alors

selon la nature de $t[m]$, par exemple, si c'est un `a`, nous pouvons en déduire que s'il existe un alignement du motif recouvrant cette occurence de la lettre `a`, le décalage i pour cet alignment est alors tel que $t[m-i+1]$ doit être un `a`.



Nous calculons alors pour chaque lettre, quelle est l'indice j de la dernière occurence de cette lettre dans le motif w. Alors si le motif apparait dans t, il apparait au plus tôt à la position $i = m-j+1$. On recommence alors la recherche depuis cette position.

En pratique, sur des textes en langues naturelles par exemple, cette algorithme est très efficace, car il peut sauter très vite de grandes portions de texte (par pas de m). C'est l'algorithme à la base de `grep`.

```
1   fonction prétraitementBM(tableau[1..m] de caractères motif) : tableau[1..|Σ|] d'entiers :
2     soit décalage := tableau[1..|Σ|] initialisé à m + 1
3     pour i de 1 à m faire
4       décalage[motif[i]] ← m − i
5     retourner d
6
7   fonction rechercheBM(tableau[1..m] de caractères motif, tableau[1..n] de caractères texte) :
8             liste d'entiers =
9     soit trouvés := ref Liste.vide()
10    soit décalage := prétraitementBM(motif)
11
12    soit la fonction calculePlusLongSuffixe(entier positionFin) : entier :=
13      soit longueur := ref 0
14      tant que !longueur < m ∧ texte[!positionFin−!longueur] = motif[m−!longueur] faire
15        longueur :=!longueur + 1
16      retourner longueur

17    soit positionFin := ref m − 1
18    tant que !positionFin < n faire
19      soit longueurLue = calculePlusLongSuffixe(!positionFin)
20      si longueurLu = m alors
21        trouvés ← Liste.insère(!positionFin + 1 − m, trouvés)
22        positionFin ←!positionFin + 1
23      sinon
24        soit positionFausse :=!positionFin − longueurLue
25        positionFin ← positionFausse + décalage[texte[positionFausse]]
26    retourner trouvés
```