

Lycée



aidherbe

**Cours
d'informatique :
option**



Liste des chapitres

I	A.B.R.	1
II	Tas	19
III	Logique	37
IV	Graphes	55
V	Graphes valués	87
VI	Flots	97
VII	Langages rationnels	109
VIII	Automates	129
IX	Langages rationnels compléments	155
X	Réponses aux exercices	167

Chapitre I

A.B.R.

1	Introduction	2
2	Arbres binaires de recherche	3
2-1	Définition	3
2-2	Recherche	4
2-3	Analyse de la recherche	4
2-4	Insertion aux feuilles	6
2-5	Insertion à la racine	7
2-6	Suppression	9
3	Application : dictionnaires	12
4	Exercices	14

1 Introduction

Nous allons, dans ce chapitre, utiliser les arbres binaires ; leur principal intérêt est que chaque nœud est accessible par un chemin de longueur h au plus depuis la racine où h est la hauteur de l'arbre. De plus la hauteur peut être aussi petite que $\log_2(n)$ où n est la taille de l'arbre (le nombre de nœuds).

Il reste alors deux étapes pour utiliser pleinement cette caractéristique.

1. Comment savoir où chercher dans l'arbre ?

Il faudra ajouter une propriété supplémentaire, c'est l'objet de ce chapitre.

2. Comment assurer une hauteur qui reste de l'ordre de $\log_2(n)$?

Il faudra maintenir des arbres équilibrés : arbres AVL, arbres rouge-noir, arbres 2-3, ...

Ce sera l'objet d'un T.P.

Dans ce chapitre les nœuds porteront une variable d'un type qui peut être composé.

On supposera donnée une fonction `cle` qui associe un entier à chaque valeur.

```
cle 'a' -> bool
```

On imposera que les clés doivent être distinctes dans l'arbre. ; autrement dit la restriction de la fonction `cle` aux valeurs de l'arbre doit être injective.

Dans les exemple, les variables seront entières et on peut écrire

```
let cle x = x;;
```

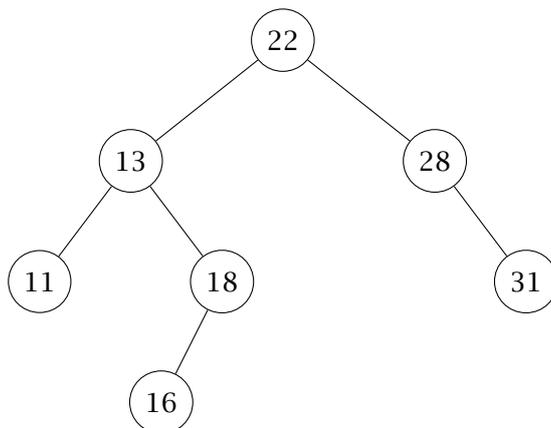
2 Arbres binaires de recherche

2-1 Définition

Arbre binaire de recherche (ABR)

Un arbre binaire de recherche est un arbre binaire tel que, pour tout nœud x , les clés des éléments du fils gauche sont strictement inférieures à la clé de x et les clés des éléments du fils droit sont strictement supérieures à la clé de x .

Exemple



La déclaration du type est la même que dans le cas d'un arbre binaire classique. Il faut vérifier soi-même que les clés sont distinctes et dans le bon ordre.

```
type 'a abr = Vide | Noeud of ('a abr*'a*'a abr);;
```

2-2 Recherche

Une fonction de base est la recherche d'un élément dans un arbre : on renvoie un booléen indiquant s'il existe un élément de clé donnée dans l'arbre. Pour cela on "descend" dans l'arbre en choisissant le fils gauche ou le fils droit selon que la valeur à chercher est inférieure ou supérieure à la clé de la racine.

- C'est le principe de la recherche par dichotomie.
- On parcourt ainsi (partiellement) une branche de l'arbre.

```
let rec chercher k arbre =  
  match arbre with  
  | Vide -> false  
  | Noeud(g, r, d) when cle r = k -> true  
  | Noeud(g, r, d) when k < cle r -> chercher k g  
  | Noeud(g, r, d) -> chercher k d;;
```

Code I.1 Recherche dans un arbre binaire de recherche

2-3 Analyse de la recherche

Nous allons établir la terminaison, la preuve et la complexité en même temps en séparant les cas où la clé recherchée est ou n'est pas dans l'arbre. La complexité est comptée en nombre de comparaisons.

Absence de la clé

Si la clé n'est pas dans l'arbre, chaque appel de la fonction à partir d'un sous-arbre non vide fera un appel à la fonction avec un des deux fils. Ainsi la hauteur de l'arbre en paramètre de la fonction diminue strictement à chaque appel. Comme la hauteur ne peut être inférieure à -1 le nombre d'appel est fini et on aboutit à un appel `chercher x Vide` qui renvoie `false`.

L'algorithme termine en renvoyant la bonne réponse. De plus la diminution d'au moins 1 de la hauteur à chaque appel récursif montre que le nombre d'appel est au plus $h + 1$. Comme on fait 2 comparaisons à chaque appel sauf pour le cas de l'arbre vide la complexité en nombre de comparaisons est majorée par $2(h + 1)$ (h est la hauteur de l'arbre).

Clé présente

En s'inspirant du résultat ci-dessus on note \mathcal{P}_h la propriété :

pour tout arbre a de hauteur h et pour toute clé k apparaissant dans l'arbre chercher k a renvoie true avec $2h + 1$ comparaisons au plus.

On notera que l'arbre est non vide car il contient au moins le nœud de clé k , ainsi $h \geq 0$.

1. \mathcal{P}_0 est vraie car un arbre de hauteur 0 qui au moins une clé ne peut être qu'un arbre de la forme $\text{Noeud}(\text{Vide}, r, \text{Vide})$ avec $\text{cle } r = k$: dans ce cas la fonction renvoie true après 1 comparaison.
2. On suppose que \mathcal{P}_m est vraie pour tout $m < h$ ($h \geq 1$).
 - a. $a = \text{Noeud}(g, r, d)$ est un arbre de hauteur h qui contient un élément de clé k .
 - i. Si on a $\text{cle } r = k$, la fonction chercher k a renvoie true avec 1 comparaison.
 - ii. Si on a $k < \text{cle } r$ alors l'élément de clé k ne peut pas être d donc g contient k et est de hauteur $h - 1$ au plus.
 chercher k a effectue 2 comparaisons puis appelle chercher k g qui, d'après l'hypothèse de récurrence, renvoie true après $2(h - 1) + 1$ comparaisons au plus.
 Ainsi chercher k a renvoie true avec $2h - 1 + 2$ comparaisons au plus.
 - iii. De même si on a $\text{cle } r > k$ alors d contient la clé k et chercher k a renvoie true avec $2h - 1 + 2$ comparaisons au plus.
3. La propriété est prouvée par récurrence.

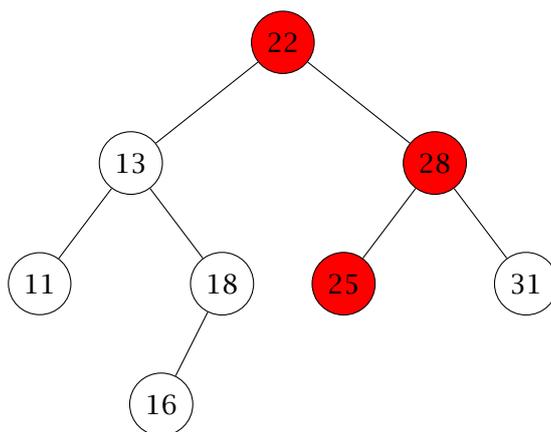
On a ainsi prouvé que l'algorithme termine dans tous les cas, qu'il renvoie la réponse exacte et que la complexité est majorée par $2h + 2$, c'est un $\mathcal{O}(h)$.

2-4 Insertion aux feuilles

On peut ajouter un nouvel élément à une feuille de l'arbre : on recherche la clé de cet élément et une recherche infructueuse arrive à un sous-arbre Vide. On peut alors y placer l'élément.

Ici encore la complexité est proportionnelle au nombre d'appels récursifs effectués donc est de la forme $\mathcal{O}(h)$ où h est la hauteur de l'arbre.

Exemple : on ajoute 25 à l'exemple initial :



```

let rec insertionF x arbre =
  match arbre with
  |Vide -> Noeud(Vide, x, Vide)
  |Noeud(g, r d) when cle r = cle x-> Noeud(g, x, d)
  |Noeud(g, r, d) when cle x = cle r
    -> Noeud((insertionF x g), r, d)
  |Noeud(g, r, d) -> Noeud(g, r, (insertionF x d));;
  
```

Code I.2 Insertion à une feuille dans un arbre binaire de recherche

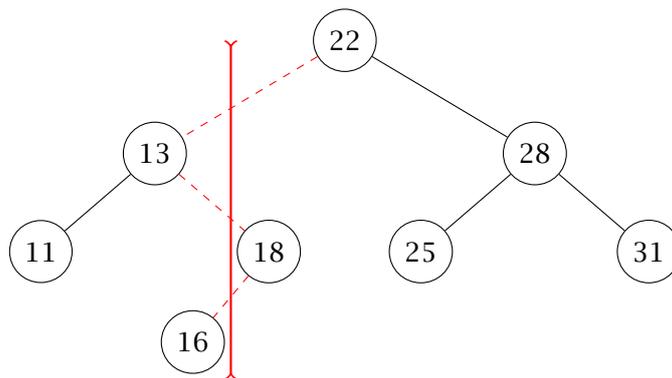
On remarquera que la valeur est changée si la clé existait déjà.

2-5 Insertion à la racine

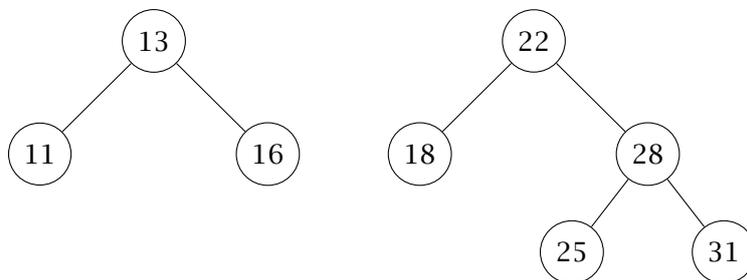
On peut aussi choisir d'insérer un nouveau nœud en le plaçant à la racine.

Pour cela on va couper l'arbre en deux parties regroupant les nœuds inférieurs à la nouvelle clé et ceux qui sont supérieurs. La récursivité permet de le faire avec un coût minimum c'est-à-dire proportionnel à la hauteur et non à la taille.

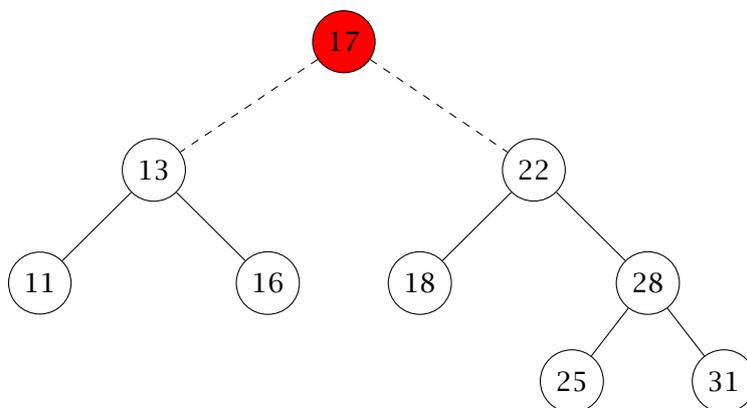
Le dernier arbre construit est coupé par 17 :



On reconstruit deux arbres :



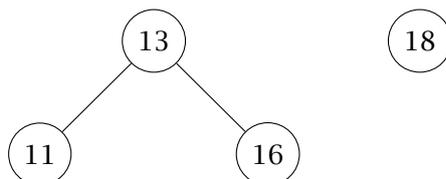
On peut alors ajouter le nœud 17 à la racine.



La première étape consiste à produire les deux arbres séparés par la clé de l'élément que l'on veut ajouter. Le découpage et la reconstitution se font en même temps, récursivement.

Si la clé de la valeur est strictement inférieure à celle de la racine.

↪ On découpe le fils gauche avec la même valeur. On obtient deux arbres : g_inf , l'arbre formés des nœuds du fils gauche de clé inférieure à la coupure et g_sup .



↪ L'arbre des nœuds inférieurs à la coupure est alors g_inf .

↪ L'arbre des nœuds supérieurs à la coupure est formé en remplaçant le fils gauche initial par g_sup .

On fait les opérations symétriques si la coupure est strictement supérieure à la racine.

En cas d'égalité on retire la racine et on renvoie les deux fils.

```

let rec decoupage k a =
  match a with
  | Vide -> (Vide,Vide)
  | Noeud(g, r, d) when cle r = r -> (g, d)
  | Noeud(g, r, d) when k < cle r
    -> let (gg, gd) = decoupage k g
        in (gg, Noeud(gd, r, d))
  | Noeud(g,r, d) -> let (dg, dd) = decoupage k d
        in (Noeud(g, r, dg),dd);;

let insertionRacine x a =
  let (g, d)= decoupage (cle x) a
  in Noeud(g,x, d);;
  
```

Code I.3 Insertion à la racine dans un arbre binaire de recherche

Ici encore on remplace l'élément de l'arbre qui a une clé égale à celle de l'élément à ajouter.

2-6 Suppression

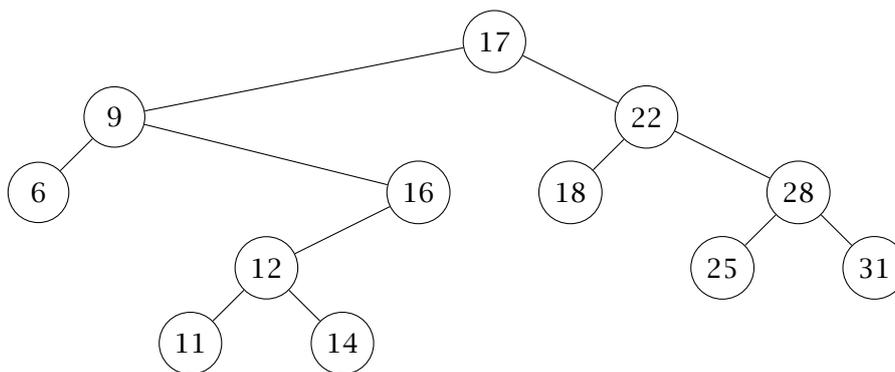
La suppression d'un nœud demande de conserver la structure ordonnée.

Il est relativement facile d'imaginer comment supprimer une feuille : on la remplace par l'arbre vide. Le cas d'un nœud interne n est plus délicat. On se ramène au cas de la racine car il suffira de remplacer le sous-arbre dont n est la racine par l'arbre diminué.

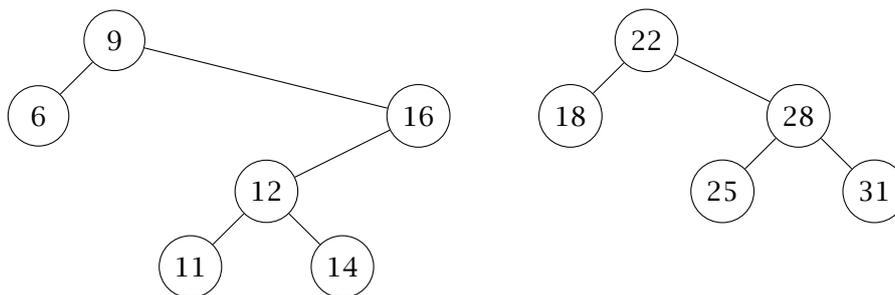
Un cas simple est celui d'un nœud sans fils gauche (ou droit) : l'arbre allégé est alors simplement le fils droit (ou gauche).

Pour ôter la racine d'un sous-arbre sans trop de modification on va choisir de le remplacer par un autre nœud de l'arbre. On choisira sa clé pour conserver la structure d'arbre binaire de recherche avec le minimum de modifications : le maximum dans le fils gauche (ou le minimum du fils droit) est un bon choix car sa clé sépare les clés restantes et on le remplace facilement par son fils gauche.

On va donc opérer en 4 étapes.

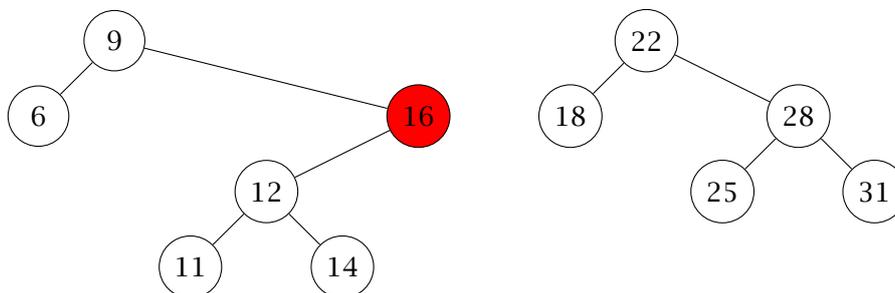


Enlever la racine



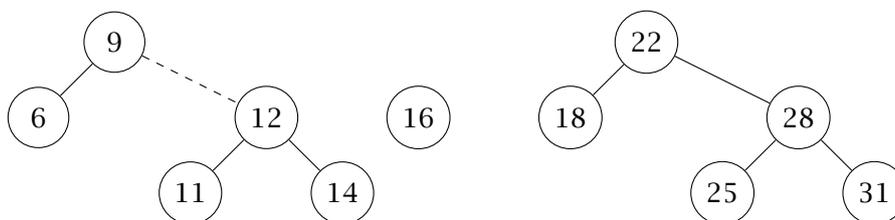
Calculer la clé maximale du fils gauche

On cherche récursivement le fils droit sans fils droit.

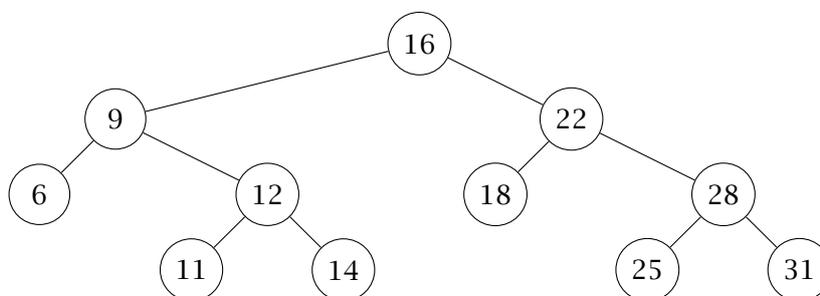


Reconstituer le fils gauche

Si T est le sous-arbre du fils gauche dont la clé est maximale alors il n'a pas de fils droit (qui, sinon, aurait des clés supérieures) ; le fils gauche de T remplace T .



Reconstruire l'arbre



On peut écrire les fonctions correspondantes

```
let rec maxArbre arbre =
  match arbre with
  |Vide -> failwith "Arbre vide"
  |Noeud(g, r, Vide) -> r
  |Noeud(g, r, d) -> maxArbre d;;
```

```

let rec suppressionMax arbre =
  match arbre with
  |Vide -> failwith "Arbre vide"
  |Noeud(g, r, Vide) -> g
  |Noeud(g, r, d) -> Noeud(g, r, suppressionMax d);;

```

```

let suppressionRacine arbre =
  match arbre with
  |Vide -> Vide
  |Noeud(Vide, r, d) -> d
  |Noeud(g, r, d) -> let y = maxArbre g in
                     Noeud(suppressionMax g, y, d);;

```

Code I.4 Suppression de la racine dans un arbre binaire de recherche

On peut alors revenir au cas général.

```

let rec suppression k arbre =
  match arbre with
  |Vide -> Vide
  |Noeud(g, r, d) when c1e r = k
                    -> suppressionRacine arbre
  |Noeud(g, r, d) when k < c1e r
                    -> Noeud((suppression k g), r, d)
  |Noeud(g, r, d) -> Noeud(g, r, (suppression k d));;

```

Code I.5 Suppression d'un nœud dans un arbre binaire de recherche

3 Application : dictionnaires

Lors de l'étude des structures abstraites nous avons définis les dictionnaires (ou tableaux associatifs) qui sont des structures de données abstraites où l'on stocke des éléments munis d'une clé.

On travaillera donc avec des éléments d'un type 'a pour lesquels la fonction `cle : 'a -> int` qui associe un indice unique à chaque élément est **injective**.

Le type abstrait doit définir les fonctions :

- ↪ `dictionnaireVide : unit -> 'a dict`, qui construit un dictionnaire vide,
- ↪ `insertion : 'a -> 'a dict -> 'a dict`, qui insère un élément dans le dictionnaire,
- ↪ `defini : int -> 'a dict -> bool`, qui teste si une clé correspond à une entrée,
- ↪ `valeur : int -> 'a dict -> 'a`, qui retrouve, s'il existe, l'élément du dictionnaire correspondant à une clé donnée.
- ↪ `retrait : int -> 'a dict -> 'a dict`, qui élimine un élément défini par sa clé.

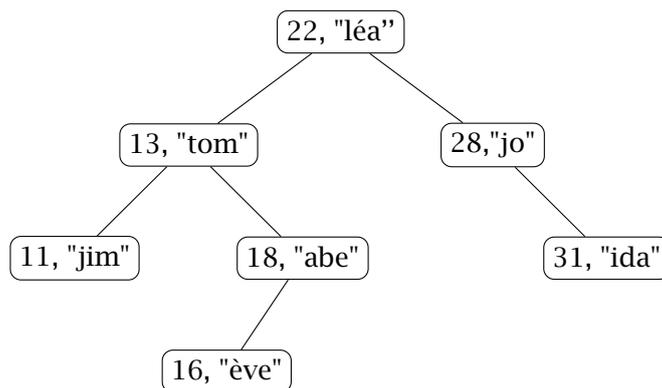
On remarquera que les fonctions ci-dessus correspondent à un type de données persistant : on conserve les dictionnaires et l'ajout ou le retrait d'un élément crée un nouveau dictionnaire.

Nous avons utilisé des tables de hachages pour implémenter cette structure : les différentes fonctions sont alors très efficaces. Cependant il est difficile dans ce cas d'énumérer les éléments, ou ceux dont les clés sont dans un intervalle. De plus le type n'est pas persistant.

On peut utiliser la structure d'arbre binaire de recherche.

Dans l'exemple les éléments seront des couples (a, b) où a est la clé.

Exemple



Les fonctions sont alors simplement des renommages.

```
let dictionnaireVide () -> Vide;;  
let insertion = insertionRacine;;  
let defini = chercher;;  
let retrait = suppression;;
```

Sauf pour la valeur d'un élément en fonction de sa clé qui est une modification simple de la recherche.

```
let rec valeur k arbre =  
  match arbre with  
  |Vide -> failwith "Il n'y a pas d'élément associé"  
  |Noeud(g, r, d) when cle r = k -> r  
  |Noeud(g, r, d) when k < cle r -> valeur k g  
  |Noeud(g, r, d) -> valeur k d;;
```

4 Exercices

Ex. 1 Une caractérisation

Prouver qu'un arbre binaire est un arbre binaire de recherche si et seulement si le parcours **infixe** des clés de l'arbre donne une suite strictement croissante.

Ex. 2 Parcours de recherche

On suppose que tous les nombres de 1 à 800 sont les clés d'un arbre binaire de recherche. Parmi les suites suivantes lesquelles ne peuvent pas être les clés d'un parcours de recherche de 417 ?

1. 222, 457, 122, 217, 417
2. 601, 403, 555, 408, 523, 411, 466, 417
3. 601, 403, 555, 408, 563, 411, 466, 417
4. 901, 403, 555, 411, 466, 417

Ex. 3 Vérification

Écrire une fonction qui teste si l'arbre binaire passé en paramètre est un arbre binaire de recherche.

Ex. 4 Tranche

Écrire une fonction `tranche arbre a b` qui reçoit comme paramètre un arbre binaire de recherche et deux entiers et qui renvoie un arbre binaire de recherche dont les clés sont les clés de l'arbre comprises entre `a` et `b`.

Ex. 5 Tri

Comment peut-on utiliser les arbres binaires de recherche pour effectuer un tri ? Si on admet que la hauteur moyenne d'un arbre binaire de recherche de taille n est un $\mathcal{O}(\log_2(n))$ déterminer la complexité, en moyenne, de ce tri.

Ex. 6 Fusion

Écrire une fonction `Fusion a b` qui joint deux arbres binaires de recherche ; on pourra utiliser les fonctions de coupure d'un arbre binaire de recherche.

Ex. 7 Ancêtre commun

Un ancêtre commun de deux nœuds n_1 et n_2 d'un arbre homogène est un nœud dont n_1 et n_2 sont descendants ; en particulier la racine est toujours un ancêtre commun de toute paire de nœuds.

1. Montrer que, pour tout entier h , deux nœuds n_1 et n_2 admettent au plus un ancêtre commun à la profondeur h .

On appelle **plus proche ancêtre commun**, noté PPAC, de n_1 et n_2 leur ancêtre commun de profondeur maximale.

D'après le résultat ci-dessus le plus proche ancêtre commun est unique.

2. Prouver que les ancêtres communs de n_1 et n_2 sont les ancêtres de leur PPAC.
3. n_1 et n_2 sont deux nœuds d'un arbre. Prouver que leur PPAC est
 1. n_1 , si n_2 est un descendant de n_1 ,
 2. n_2 , si n_1 est un descendant de n_2 ,
 3. l'unique nœud n de l'arbre tel que n_1 appartient au fils gauche (resp. fils droit) de n et n_2 appartient au fils droit (resp. fils gauche) de n dans les autres cas.
4. En déduire l'écriture d'une fonction qui permet de déterminer le PPAC de deux nœuds d'un A.B.R.

Ex. 8 Successeurs et prédécesseurs

On considère les nœuds d'un arbre binaire de recherche a

Le successeur (resp. prédécesseur) d'un nœud de clé n est le nœud de l'arbre dont la clé suit (resp. précède) n dans le parcours infixe. On remarquera que le successeur peut ne pas exister, dans le cas où la clé du nœud est maximale.

1. Montrer que si un arbre binaire de recherche a un fils gauche (resp. un fils droit) alors le prédécesseur de sa racine n'a pas de fils droit (resp. le successeur de sa racine n'a pas de fils gauche).
2. Montrer que si b est le successeur de a alors
soit b est le nœud de clé minimale dans le fils droit de a ,
soit a est le nœud de clé maximale dans le fils gauche de b .
3. Écrire un algorithme calculant la plus petite clé strictement supérieure à une valeur k donnée dans un arbre binaire de recherche. Cela permet de calculer le successeur d'un nœud de clé k .

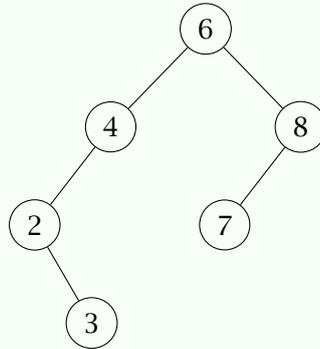
Ex. 9 Préfixes d'un arbre binaire

On dit qu'un arbre binaire t est un préfixe d'un arbre binaire $t1$ si t est l'arbre vide ou si $t = \text{Noeud}(g, k, d)$ et $t1 = \text{Noeud}(g1, k, d1)$ avec g préfixe de $g1$ et d préfixe de $d1$.

1. Montrer que la relation "*est préfixe de*" est un ordre partiel sur les arbres binaires.
2. Montrer que tout préfixe d'un ABR est un ABR.
3. Écrire une fonction testant si un arbre binaire est préfixe d'un autre.

Ex. 10 Nombre de constructions

Déterminer tous les ordres possibles d'insertion de clés à partir de l'arbre vide qui donnent l'arbre suivant par adjonction aux feuilles



Plus généralement montrer que le nombre de permutations des clés d'un arbre T , de taille n , qui engendrent l'arbre par adjonctions successives aux feuilles est $n!$ divisé par le produit des tailles de tous les sous-arbres (y compris T) de T .

Ex. 11 Complexité moyenne d'une recherche

On considère les arbres binaires de recherche construits en ajoutant à l'arbre vide les nœuds de clé 1 à n dans le désordre.

On suppose que les $n!$ ordres d'insertion sont équiprobables.

On cherche la complexité moyenne, en nombre de comparaisons, de la recherche d'une clé dans l'arbre : a_n .

1. Montrer que la probabilité que la clé de la racine soit i est $\frac{1}{n}$.

On note $a_{i,n}$ la complexité moyenne, en nombre de comparaisons, de la recherche d'une clé dans un arbre dont la racine a la clé i .

2. Prouver que $a_{i,n} = (a_{i-1} + 2) \frac{i-1}{n} + \frac{1}{n} + (a_{n-i} + 2) \frac{n-i}{n}$.

3. En déduire que $a_n = 2 - \frac{1}{n} + \frac{2}{n^2} \sum_{i=1}^{n-1} i a_i$

4. En calculant $n^2 a_n - (n-1)^2 a_{n-1}$ prouver que $a_n = \frac{1}{n^2} ((n^2 - 1) a_{n-1} + 4n - 3)$

5. Prouver que $a_n = \mathcal{O}(\log_2(n))$, on pourra poser $u_n = \frac{n}{n+1} a_n$.

Chapitre II

Tas

1	Files de priorité	20
1-1	Structure abstraite	20
1-2	Réalisation par A.B.R.	21
2	Tas	23
2-1	Définitions	23
2-2	Opérations abstraites	25
2-3	Réalisation par tableaux	27
3	Exercices	30
3-1	Heapsort	30
3-2	Tas modifiables	32
3-3	Tableaux fonctionnels	33
3-4	Arbres gauchistes	35

1 Files de priorité

1-1 Structure abstraite

Dans ce chapitre nous allons définir et implémenter un type de données efficace pour traduire le type abstrait de file de priorité.

On rappelle qu'il s'agit de maintenir un ensemble d'objets d'un type fixé accompagnés d'une fonction `priorite` donnant la priorité.

```
priorite : 'a -> int = <fun>
```

On veut pouvoir ajouter un élément et exhiber ou retirer l'élément de priorité maximale.

On souhaite alors définir un type de donnée `'a fdp` de file de priorité dont les objets sont de type `'a` ainsi que les fonctions

```
fpVide : 'a -> 'a fdp = <fun>
estVide : 'a fdp -> bool = <fun>
ajouter : 'a -> 'a fdp -> unit() = <fun>
premier : 'a fdp -> 'a = <fun>
enlever : 'a fdp -> unit() = <fun>
```

↪ `fpVide` demande un élément de type `'a` qui sera l'élément par défaut lorsque l'on aura besoin de tableaux.

↪ `ajouter` modifie une file de priorité en lui ajoutant un élément,

↪ `premier` renvoie l'élément de priorité maximale dans une file de priorité sans la modifier.

↪ `enlever` retire l'élément de priorité maximale d'une file de priorité.

On remarque qu'on choisit une structure mutable.

Bien entendu on souhaite une structure qui permette des fonctions dont les temps de calculs seront les plus petits possibles et dont l'occupation en mémoire est faible aussi.

Rappels

Il y a quatre choix pour l'implémentation avec les structures de données classiques

1. on peut choisir des listes ou des tableaux
2. pour chaque structure on peut soit trier lors de l'adjonction soit extraire le maximum.

Dans le cas de structures triées avec n éléments `ajoute` est de complexité en $\mathcal{O}(n)$ et `premier` et `reste` sont de complexité en $\mathcal{O}(1)$.

Dans le cas de structures non triées avec n éléments `ajoute` est de complexité en $\mathcal{O}(1)$ et `premier` et `reste` sont de complexité en $\mathcal{O}(n)$.

1-2 Réalisation par A.B.R.

Nous avons vu une structure qui maintient l'ordre des clés : les arbres binaires de recherche. Un arbre binaire de recherche est une bonne structure pour maintenir un ensemble d'objets ordonnables. On peut donc imaginer l'utiliser pour implémenter une file de priorité.

Toutes les opérations sont déjà écrites.

```

type 'a fdp = {mutable contenu : 'a abr};;

let cle = priorite;;

let fpVide a = {contenu = Vide};;

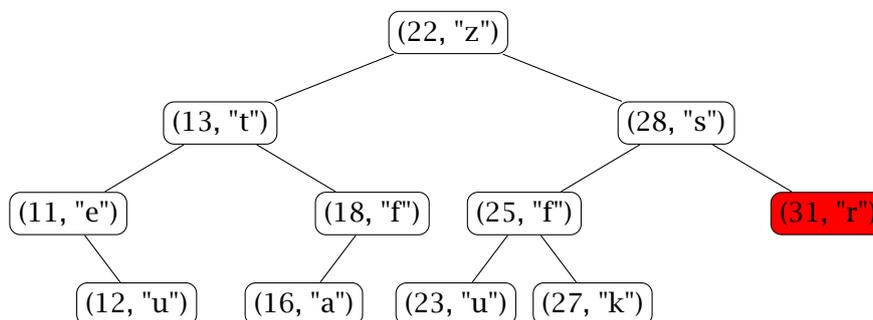
let estVide fp = fp.contenu = Vide;;

let ajouter x k fp =
  fp.contenu <- ajouterRacine (x, k) fp.contenu;;

```

Il reste cependant à gérer la recherche et le retrait de l'élément minimal car on n'en connaît pas la clé.

On remarque cependant que la structure d'arbre binaire de recherche implique que l'élément de priorité maximale est la racine du dernier fils droit non vide.



N.B. Dans les représentations graphiques on simplifiera l'écriture en choisissant les entiers pour le type 'a et la priorité sera la valeur : `let priorite x = x;;`

On peut donc écrire

```
let premier fp =
  let rec aux arbre =
    match arbre with
    |Vide -> failwith "La file est vide"
    |Noeud(_, n, Vide) -> n
    |Noeud(_, _, d) -> aux d in
  aux fp.contenu;;
```

```
let enlever fp =
  let rec aux arbre =
    match arbre with
    |Vide -> failwith "La file est vide"
    |Noeud(g, _, Vide) -> g
    |Noeud(g, n, d) -> Noeud(g, n, aux d) in
  fp.contenu <- aux fp.contenu;;
```

La complexité de ajouter, premier et enlever est de l'ordre de la hauteur de l'arbre : en moyenne on peut compter sur une complexité en $\mathcal{O}(\log_2(n))$ mais elle peut atteindre n dans le pire des cas si on ne maintient pas un arbre équilibré. La structure d'arbre est persistante, cela implique que la complexité spatiale peut devenir importante : à chaque ajout et retrait on crée un nombre de nœuds de l'ordre de la hauteur.

2 Tas

Nous allons maintenant définir une structure d'arbre moins structurée que celle d'arbre binaire de recherche mais suffisante pour gérer les priorités.

2-1 Définitions

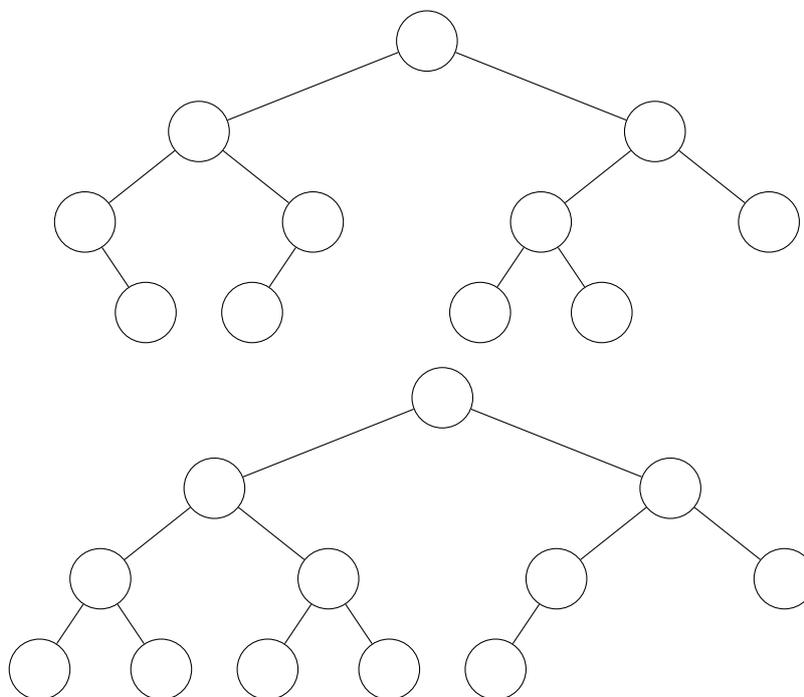
Quelques rappels

Arbres quasi-complets

Un arbre est quasi-complet si, en notant h sa hauteur toutes les feuilles sont de profondeur h ou $h - 1$ et si le nombre de nœuds de profondeur $h - 1$ est 2^{h-1} .

Un arbre complet est quasi-complet à gauche si, de plus, les nœuds de profondeur h sont tous situés à gauche.

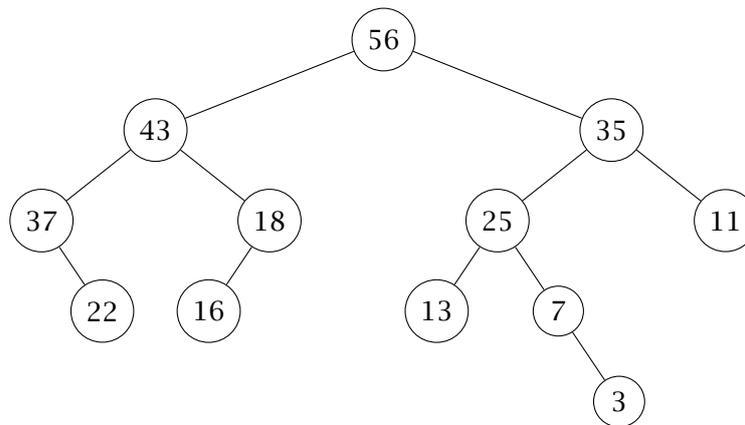
On a vu qu'alors le nombre de nœuds de profondeur k est 2^k pour tout $k < h$.
On en déduit que la taille n de l'arbre vérifie $2^{h-1} \leq n < 2^h$.



Arbres décroissants

Un arbre décroissant est un arbre binaire dont les nœuds possèdent une clé et tel que la clé d'un nœud est toujours inférieure à celle de son père.

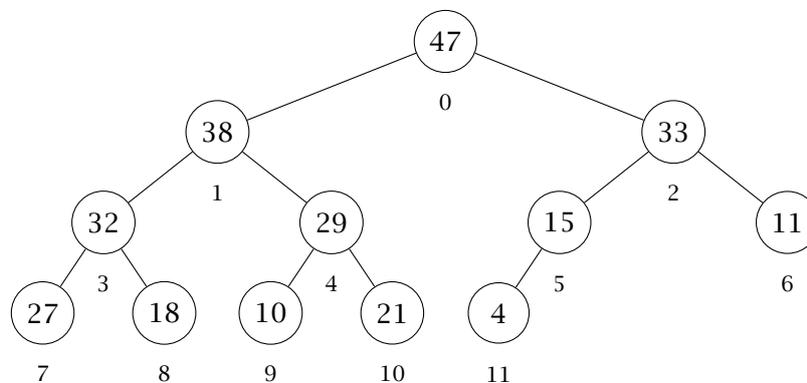
On a donc remplacé la croissance des clé "en largeur" des arbres binaires de recherche par une croissance du bas vers le haut moins contraignante.



Tas

Un tas est un arbre décroissant quasi-complet à gauche.

La quasi-complétude à gauche permet de numéroter les nœuds de 0 à $n - 1$ en fonction de leur place dans le parcours en largeur.



2-2 Opérations abstraites

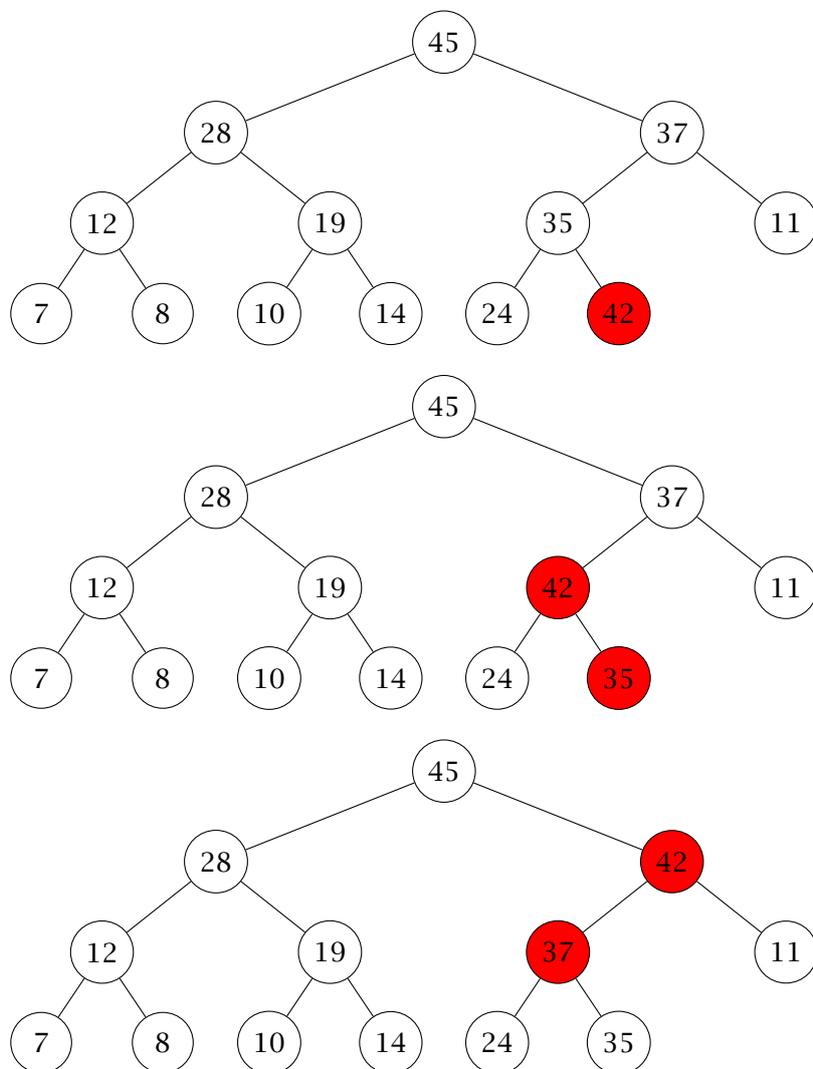
Un tas est donc un arbre dans lequel l'élément de clé maximale est à la racine et qui est fortement équilibré.

En particulier la fonction premier renvoie la racine.

Ajout

Pour ajouter un élément commence par l'ajouter à la position suivante dans l'ordre du parcours en largeur puis on le remonte tant que sa valeur est supérieure à celle de son père.

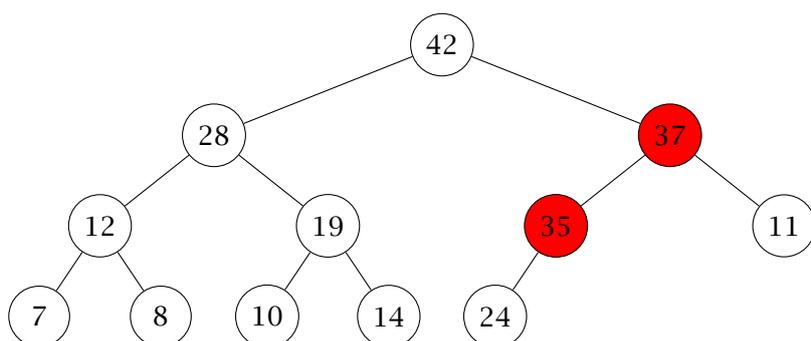
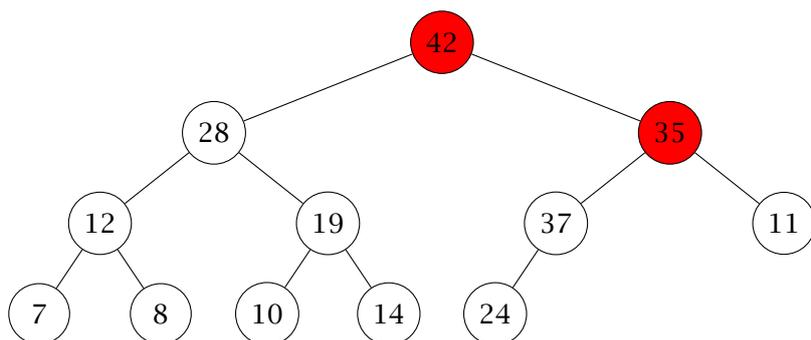
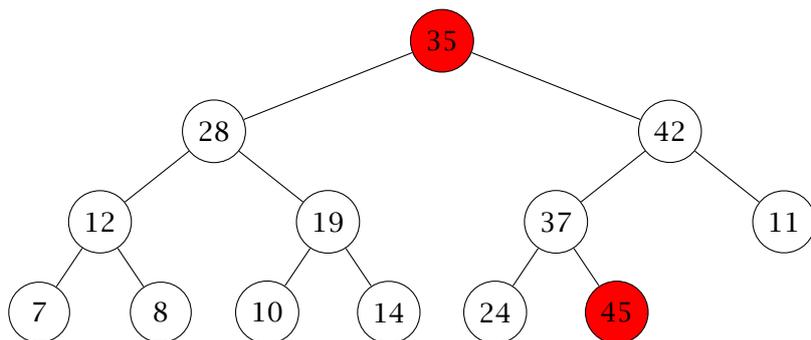
Dans l'exemple on ajoute un élément de priorité 42.



Enlever

Pour ôter la racine on peut

- ↪ échanger la racine avec le dernier élément,
- ↪ enlever le dernier élément
- ↪ à partir de la racine, échanger l'élément de la racine avec le plus grand de ses fils jusqu'à ce qu'on obtienne un tas

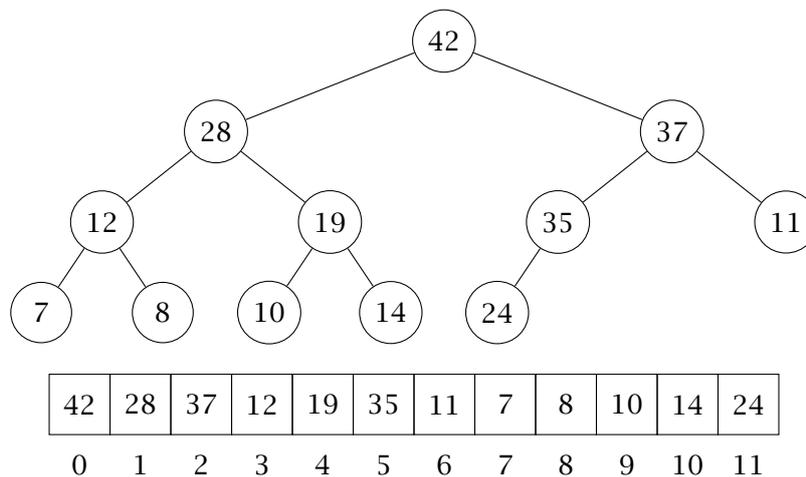


2-3 Réalisation par tableaux

Les constructions ci-dessus ne sont pas directement réalisables. En effet on doit avoir accès au nœud le plus à gauche dans la dernière profondeur ou au premier nœud libre. La numérotation par le parcours transverse permet de maintenir le numéro de ces nœuds mais le chemin qui permet d'y arriver n'est pas déterminable simplement (voir l'exercice sur les tableaux fonctionnels pour une modification). De plus on doit parfois parcourir l'arbre depuis les feuilles et la structure usuelle ne permet pas le calcul du père.

Le parcours en largeur d'un arbre le transforme en fait en tableau. Comme la structure de tableau n'est pas dynamique, le nombre d'éléments est fixé, on va donc définir une taille assez grande pour pouvoir contenir les files de taille maximale. Bien entendu il est difficile de prévoir et il y a un risque de débordement.

On doit alors maintenir un entier qui sera la taille de la file.



```
let nMax = 100;;
```

```
type 'a tas = {mutable taille : int;
               arbre : 'a array};;
```

```
let fpVide a =
  {taille = 0; arbre = Array.make nMax a};;
```

On peut facilement tester le vide et trouver l'élément minimal :

```
let estVide t =
  t.taille = 0;;
```

```
let premier t =
  t.arbre.(0);;
```

On aura besoin de la traditionnelle fonction d'échange :

```
let echange tas i j =
  let t = tas.arbre in
  let temp = t.(i) in
  t.(i) <- t.(j);
  t.(j) <- temp;;
```

Pour les fonctions `ajouter` et `enlever` on a besoin de connaître l'indice du père d'un nœud ou les indices de ses fils : on remarque que, avec la numérotation choisie, le père du nœud d'indice i a pour indice $(i - 1)/2$ et ses fils ont pour indices $2i + 1$ et $2i + 2$ respectivement.

L'ajout se fait comme indiqué ci-dessus avec une fonction auxiliaire pour effectuer la remontée.

```
let rec remonter n tas =
  let t = tas.arbre in
  if n > 0
  then let m = (n-1)/2 in
        if (priorite t.(m)) < (priorite t.(n))
        then (echange tas m n; remonter m tas);;

let ajouter x tas =
  let n = tas.taille in
  tas.arbre.(n) <- x;
  tas.taille <- n + 1;
  remonter n tas;;
```

Code II.1 Ajout d'un élément dans un tas

Pour la fonction `enlever` on a vu qu'il redescend les éléments et effectuant un échange avec le fils de priorité maximale.

On commence par rechercher ce fils. Comme il peut ne pas exister de fils on va renvoyer un type optionnel (`Some/None`) qui permet de gérer l'absence de réponse. Le dernier indice possible est $n - 1$ si n est la taille, un nœud d'indice k peut admettre des fils d'indice $2k + 1$ et $2k + 2$.

Le nombre de fils est donc 0 pour $n - 1 < 2k + 1$, 1 pour $n - 1 = 2k + 1$ et 2 pour $n - 1 > 2k + 1$.

```

let filsGrand k tas =
  let t = tas.arbre and n = tas.taille in
  if n <= 2*k + 1
  then None
  else if n = 2*k + 2
    then Some (2*k + 1)
    else if (priorite t.(2*k+1)) < (priorite t.(2*k+2))
      then Some (2*k + 2) else Some (2*k + 1);;

```

Code II.2 Recherche du fils prioritaire

```

let rec descendre k tas =
  let t = tas.arbre in
  match filsGrand k tas with
  |Some p when (priorite t.(p)) > (priorite t.(k))
    -> (echange tas k p; descendre p tas)
  |_ -> ();;

let enlever tas =
  let n = tas.taille -1 in
  echange tas 0 n;
  tas.taille <- n;
  descendre 0 tas;;

```

Code II.3 Retrait du terme de priorité maximale dans un tas

Les complexités des fonctions `ajouter` et `enlever` sont celles de `monter` et `descendre`. Dans ces dernières on fait un nombre fini d'opérations et on appelle récursivement avec un nœud de profondeur modifiée de 1 ou -1 donc les complexités sont en $\mathcal{O}(h)$ où h est la hauteur. Comme l'arbre est équilibré on en déduit

Complexités dans un tas

Les complexités des fonctions `ajouter` et `enlever` sont en $\mathcal{O}(\ln(n))$ où n est la taille du tas. Les autres fonctions sont de complexité constante.

3 Exercices

Ex. 1 Test de tas Écrire une fonction qui teste si un tableau correspond à un tas.

Ex. 2 Suppression dans un tas Écrire une fonction qui supprime, dans un tas, l'élément d'indice k .

3-1 Heapsort

Dès qu'on a une structure de file de priorité on peut l'utiliser pour réaliser un tri.

On remplit une file à partir d'une file vide vide en ajoutant un à un les éléments d'un tableau (ou une liste). On retire ensuite les éléments un à un en formant un tableau : celui-ci sera trié.

La complexité du tri est alors un $\mathcal{O}(n(a + b))$ où a et b sont les complexité respective de l'ajout et du retrait d'un élément.

Comme on sait que la complexité d'un tri par comparaisons est au moins un $\mathcal{O}(n \log(n))$ on voit qu'une au moins des deux opérations ci-dessus doit être en $\mathcal{O}(\log(n))$.

On peut considérer que le tri par insertion correspond à ce schéma avec une file de priorité implémentée par un tableau trié tandis que tri par sélection correspond à une file de priorité implémentée par un tableau non trié. On retrouve bien une complexité en $\mathcal{O}(n^2)$ car l'une des deux opérations est de complexité linéaire.

L'implémentation par un tas donne un tri de complexité asymptotique optimale.

Comme dans le cas des tris par insertions on ne va pas écrire le tri en ne se servant que des fonction du type abstrait : on va travailler directement sur la structure de données.

Si t est un tableau, il peut être vu comme un arbre quasi-complet à gauche mais cet arbre n'est pas, en général, décroissant. Comme les éléments sont déjà présents dans le tableau on peut le transformer en tas en remontant les éléments depuis l'indice 0 vers l'indice $n - 1$

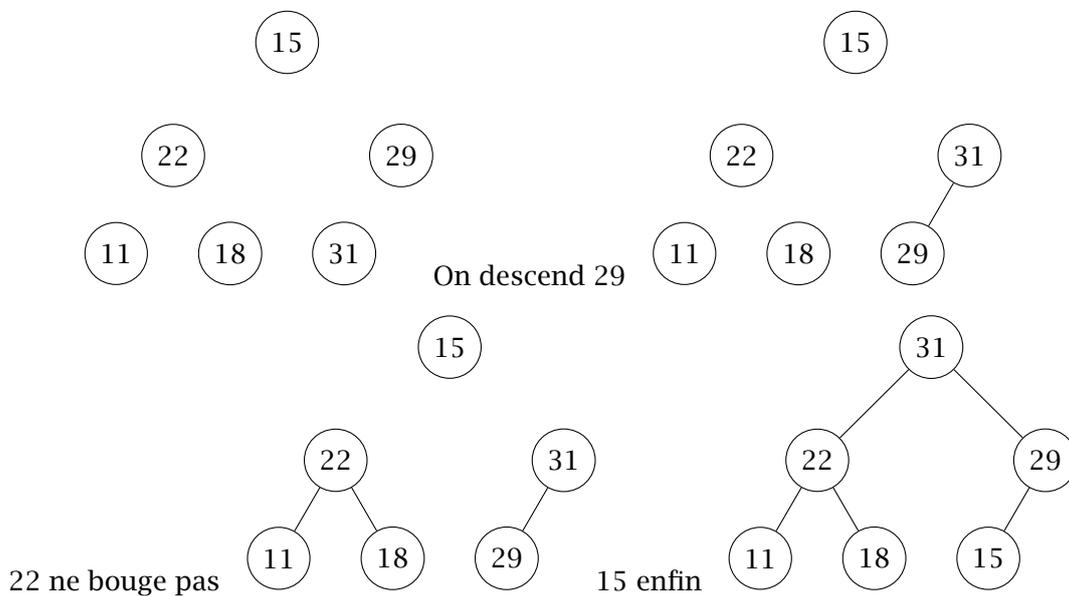
Ex. 3 Création d'un tas En utilisant uniquement la fonction remonter, écrire une fonction qui transforme t en un tas.
Quelle en est la complexité ?

Lorsqu'on en enlève tous les éléments d'un tas un par un ceux-ci sont placés dans les emplacements depuis la position $n - 1$ vers la gauche. Les n premier termes du tableau sont alors triés car le premier élément retiré est celui de priorité maximale.

Ex. 4 Tri d'un tas En vous inspirant de cette remarque écrire une fonction (sans résultat) qui trie un tableau en place.

La première étape peut être améliorée en construisant le tas non pas du haut vers le bas mais en construisant des tas depuis le bas. Initialement les nœuds forment des tas de taille 1 juxtaposés. On va former des tas de plus en plus gros en descendant les pères pour former un tas avec leurs fils qui eux-mêmes des tas. Pour cela on doit travailler depuis le dernier père (dans l'ordre des indices) vers le premier.

Dans l'exemple suivant on part de [15; 22; 29; 11; 18; 31] et on n'indique que les liaisons qui font un tas



Ex. 5 Une amélioration

Écrire une fonction qui transforme `t` en un tas avec cette méthode
 Quelle est sa complexité ?

3-2 Tas modifiables

Nous verrons dans certains algorithmes qu'il peut être utile de modifier la priorité des objets mis dans la file d'attente.

Nous supposons que les objets de la file d'attente sont de type `'a` et sont munis de fonctions qui permettent, respectivement, de calculer la priorité, de changer cette priorité et de définir un indice (unique) pour chaque objet.

Il est, par exemple, possible, de gérer les différents indices à l'aide d'un dictionnaire.

Nous supposons, pour simplifier, que les indices sont des entiers majorés **a-priori** par une valeur `nMax` qui sera définie comme une variable globale. On gèrera les indices dans un tableau. La taille de la file est alors aussi majorée par cet entier. Cette simplification se produira dans l'emploi des tas modifiables dans le cas d'algorithmes sur les graphes.

Les fonctions seront donc de la forme

```
priorite : 'a -> int = <fun>

newPrio : 'a -> int -> 'a = <fun>

indice : 'a -> int = <fun>
```

Le tableau des indices donnera la position dans le tableau associé à l'arbre.

On définit ainsi un tas modifiable par le type

```
type 'a tas = {mutable taille : int;
               arbre : 'a array;
               position : int array};;
```

`position` devra permettre le calcul de la réciproque de `indice` :

`t.position.(indice (t.arbre.(i)))` renvoie `i` et

`indice (t.arbre.(t.position.(k)))` renvoie `k`.

La position sera notée `-1` si l'objet n'est pas dans la file d'attente.

Ex. 6 Tas modifiables

Modifier les fonctions définies dans le cours et ajouter des fonctions
 present index tas qui teste si un indice est déjà présent et
 changerPrio index valeur tas qui modifie la clé de priorité avec la valeur
 indiquée.

3-3 Tableaux fonctionnels

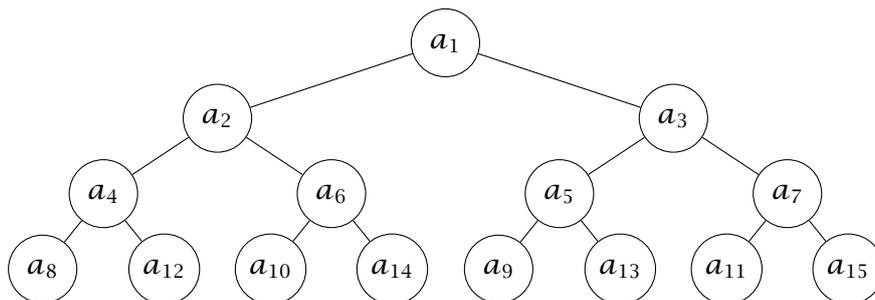
On a vu qu'on pouvait réaliser un arbre (complet à gauche) par un tableau.

Nous nous intéressons ici au problème inverse : donner une implémentation d'un tableau sous la forme d'un arbre (persistant).

Un tableau peut être vu comme une fonction depuis un ensemble d'indices vers l'ensemble des valeurs. On considère la fonction suivante qui associe un nœud à un entier non nul.

```
let rec elt k arbre =
  match arbre with
  | Vide -> failwith "Element non defini"
  | Noeud(g,n,d) -> if k = 1 then n
                    else if k mod 2 = 0
                        then elt (k/2) g else elt (k/2) d;;
```

Dans le cas $n = 15$ on accède ainsi aux nœuds de l'arbre complet de hauteur 3.

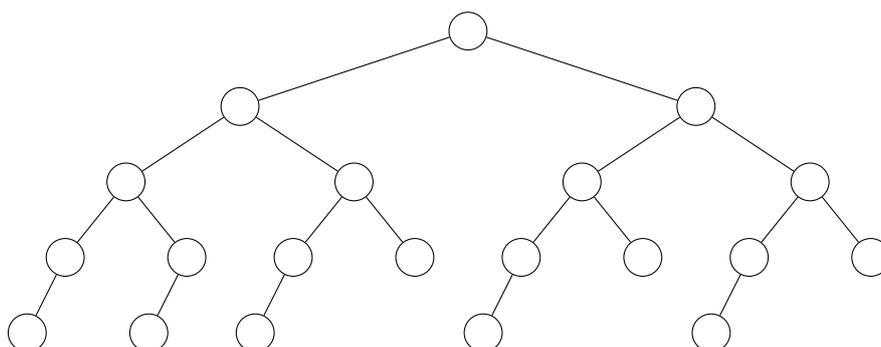


On a ainsi défini une structure de tableau à accès presque constant qui a l'avantage d'être persistante (quand on change une valeur, l'ancien tableau peut exister encore) et qui, de plus, peut-être augmentée.

Ex. 7 Tableaux fonctionnels

1. Écrire une fonction `change k x a` qui renvoie un nouvel arbre a' tel que $a'_k = x$ et $a'_i = a_i$ pour $i \neq k$. Si la position n'était pas occupée mais si le père existe on ajoutera l'élément à sa place.
2. Écrire une fonction `creerTableau n x` qui renvoie un arbre de taille n dont tous les éléments ont la valeur x .
3. Quelle est la complexité de `elt` et de `change` pour un tableau de taille n ?

On conserve la structure ci-dessus : on considère des arbres binaires de taille n tels que les nœuds occupés correspondent aux indices de 1 à n . Ces arbres correspondent donc des tableaux de taille n : on nommera **arbres de Braun** de tels arbres.



Ex. 8 Structure des arbres de Braun $t = \text{Noeud}(g, x, d)$ est un arbre de Braun.

1. Montrer que g et d sont des arbres de Braun.
2. Montrer que $\text{taille } g = \text{taille } d$ ou $\text{taille } g = \text{taille } d + 1$.
3. Montrer que $\text{hauteur } g = \text{hauteur } d$ ou $\text{hauteur } g = \text{hauteur } d + 1$.
4. Montrer que la feuille la plus à gauche est à la profondeur maximale.

On définit la fonction

```

let rec ajouterBraun x t =
  match t with
  | Vide -> Noeud(Vide,x,Vide)
  | Noeud(g,y,d) -> Noeud(ajouterBraun x d,y,g);;

```

Ex. 9 Modification des arbres de Braun

1. Prouver que si t est un arbre de Braun alors $\text{ajouter } x \ t$ est encore un arbre de Braun.
2. Comment enlever un élément en conservant la structure d'arbre de Braun ?

Un **tas de Braun** est un arbre de Braun décroissant.

Ex. 10 Tas de Braun

1. Modifier la fonction `ajouterBraun` en préservant la structure de tas.
2. Comment enlever le plus petit élément en conservant la structure de tas de Braun ?

3-4 Arbres gauchistes

Nous avons implémenté les files d'attentes avec la structure de tas. Nous avons traduit la structure arborescente en tableau ce qui nous a fait perdre la persistance. Nous allons proposer une autre structure, qui restera récursive, permettant d'implémenter une file d'attente avec des opérations de complexité garantie en $O(\log_2(n))$.

Dans un arbre le **rang** d'un nœud est la longueur de la plus courte branche (entre la racine et une feuille) où on compte les nœuds. C'est la plus petite profondeur des fils vides. Dans un tas le rang est h où h est la hauteur sauf s'il est parfait, dans ce cas le rang est $h + 1$.

On implémentera le type d'arbre binaire avec le rang :

```

type 'a arbreR = Vide | Noeud of int * 'a arbreR * 'a * 'a arbreR;;

```

Un arbre avec un seul élément est de rang 1 : `Noeud(1, Vide, x, Vide)`

Un arbre est **gauchiste** si, pour tout nœud, le rang du fils gauche est supérieur au rang du fils droit.

Ex. 11 Arbres gauchistes

1. Prouver qu'un arbre gauchiste de rang r admet au moins $2^r - 1$ nœuds.
2. Prouver que, dans un arbre gauchiste de rang r , le chemin qui choisit le fils droit à chaque nœud est longueur r (en nombre de nœuds).
3. Montrer qu'on peut fusionner deux arbres gauchistes **décroissants** en un arbre gauchiste décroissant avec une complexité en $O(r_1 + r_2)$ où r_1 et r_2 sont les rangs des arbres.
4. En déduire une implémentation persistante des files de priorité.

Chapitre **III**

Logique

1	Introduction	38
2	Syntaxe	39
2-1	Formules bien formées	39
2-2	Définition par induction	40
2-3	Simplification	41
2-4	Arbre syntactique	42
3	Sémantique	43
4	Équivalence	45
4-1	Définition	45
4-2	Formes normales	47
5	Exercices	49

Résumé

La logique est l'outil qui a été créé pour étudier et formaliser le raisonnement mathématique. Elle s'est par la suite imposée comme une partie essentielle de l'informatique. Une caractéristique principale de la logique est qu'elle fait apparaître nettement la différence entre la syntaxe, qui définit les règles formelles de manipulation des symboles utilisés, et la sémantique, qui étudie la signification ou l'interprétation des symboles utilisés.

1 Introduction

La logique étudie les énoncés qui sont susceptibles d'être vrais ou faux.

Proposition

Une proposition est un énoncé de quelque nature que ce soit qui peut être qualifié de vrai ou faux.

Dans le vocabulaire informatique on dira plutôt expression booléenne.

Ces énoncés peuvent être simples : $1 + 3 = 5$, $x^2 - 3x + 2 = 0$ admet 2 comme solution, "il fait beau", ...

Mais ils peuvent être composés d'un assemblage : "si le Groenland est recouvert de sucre glace alors $2+2=5$ ", "le ciel est bleu et les oiseaux chantent", ...

Il existe des énoncés qui ne sont pas des proposition : les phrases impératives ou interrogatives par exemple. On peut aussi penser aux énoncés qui ne peuvent avoir de sens, soit parce qu'ils emploient des mots non définis, "les duramiles sont blouvus", soit parce qu'il est impossible de leur donner une vérité (en général à cause d'une auto-référence), "cet énoncé est faux".

2 Syntaxe

2-1 Formules bien formées

Nous allons représenter les énoncés simples par des **variables propositionnelles**. Dans la pratique nous n'utiliserons qu'un nombre fini de telles variables mais nous supposerons que l'ensemble de ces variables peut être dénombrable.

On notera \mathcal{A} cet ensemble ; ses éléments pourront être notés p, q, r, \dots ou $p_0, p_1, p_2 \dots$

Si A est un ensemble on note A^* l'ensemble des assemblages (les **mots**) d'éléments de A (les **lettres**).

Ils sont représentés en collant les lettres : $w = x_1x_2\dots x_m$.

Si \mathcal{A} est un ensemble de variables propositionnelles, on note $\mathcal{A}' = \mathcal{A} \cup \{ (,), \neg, \wedge, \vee \}$.

Les ensembles considérés sont des sous-ensembles de $(\mathcal{A}')^*$.

Formules

L'ensemble des formules (bien formées) de la logique des propositions sur \mathcal{A} est le plus petit ensemble \mathcal{L} (ou $\mathcal{L}_{\mathcal{A}}$) tel que

1. toute variable propositionnelle (élément de \mathcal{A}) appartient à \mathcal{L} ,
2. si $F \in \mathcal{L}$ est une formule, alors $(\neg F) \in \mathcal{L}$,
3. si $F \in \mathcal{L}$ et $G \in \mathcal{L}$ alors $(F \wedge G) \in \mathcal{L}$,
4. si $F \in \mathcal{L}$ et $G \in \mathcal{L}$ alors $(F \vee G) \in \mathcal{L}$.

On note \mathbb{L} l'ensemble des ensembles qui vérifient les propriétés 1. à 4.

Cette définition d'apparence anodine contient plusieurs présupposés :

- a. il existe au moins un ensemble qui vérifie ces propriétés,
- b. on a une notion de "plus petit",
- c. il existe un plus petit ensemble.

Voici le sens que l'on donne à ces phrases.

- a. Un ensemble vérifiant les propriétés existe : $(\mathcal{A}')^*$ est un exemple.
- b. Plus petit est défini par l'inclusion. Un élément minimal dans un ensemble d'ensembles \mathbb{L} est alors inclus dans tous les éléments de \mathbb{L} . L'unicité d'un éventuel plus petit ensemble est alors

acquise car, si L_1 et L_2 sont inclus dans tous les ensembles de \mathbb{L} , alors $L_1 \subset L_2$ et $L_2 \subset L_1$ donc $L_1 = L_2$.

c. On définit $\mathcal{L} = \bigcap_{L \in \mathbb{L}} L$: montrons que \mathcal{L} répond à toutes nos exigences.

1. On a $\mathcal{A} \subset L$ pour tout L donc $\mathcal{A} \subset \mathcal{L}$,
2. si F appartient à \mathcal{L} alors il appartient à tous les ensembles L donc $(\neg F)$ appartient à tous les ensembles L ; on en déduit que $(\neg F) \in \mathcal{L}$,
3. si F et G appartiennent à \mathcal{L} alors ils appartiennent à tous les ensembles $L \in \mathbb{L}$ donc $(F \wedge G) \in L$ pour tout L d'où $(F \wedge G) \in \mathcal{L}$,
4. de même $(F \vee G) \in \mathcal{L}$ pour tous F et G appartenant à \mathcal{L} .

Ainsi \mathcal{L} vérifie les propriétés de la définition.

De plus, si L vérifie les propriétés, alors $L \in \mathbb{L}$ donc, par construction, $\mathcal{L} \subset L$: \mathcal{L} est plus petit que tous les ensembles de \mathbb{L} .

2-2 Définition par induction

Cette construction de \mathcal{L} ne permet pas facilement de reconnaître les formules.

Voici une autre manière de définir l'ensemble des formules.

↪ On définit $\mathcal{L}_0 = \mathcal{A}$.

↪ Si \mathcal{L}_n est défini on construit \mathcal{L}_{n+1} par

$$\mathcal{L}_{n+1} = \mathcal{L}_n \cup \{\neg F, F \in \mathcal{L}_n\} \cup \{F \wedge G, F, G \in \mathcal{L}_n\} \cup \{F \vee G, F, G \in \mathcal{L}_n\}$$

Définition inductive

$$\mathcal{L} = \bigcup_{n \in \mathbb{N}} \mathcal{L}_n$$

Démonstration On note $\mathcal{L}_\infty = \bigcup_{n \in \mathbb{N}} \mathcal{L}_n$.

On remarque que $\mathcal{L}_n \subset \mathcal{L}_{n+1}$ donc $\mathcal{L}_p \subset \mathcal{L}_q$ pour $p \leq q$.

1. On a $\mathcal{A} = \mathcal{L}_0 \subset \mathcal{L}_\infty$.

2. Si F appartient à \mathcal{L}_∞ alors il existe $n \in \mathbb{N}$ tel que $F \in \mathcal{L}_n$.

On a alors $\neg F \in \mathcal{L}_{n+1}$ d'où $\neg F \in \mathcal{L}_\infty$.

3. Si F et G appartiennent à \mathcal{L}_∞ alors il existe $p, q \in \mathbb{N}$ tel que $F \in \mathcal{L}_p$ et $G \in \mathcal{L}_q$.

Si on pose $n = \max\{p, q\}$, on a $F \in \mathcal{L}_p \subset \mathcal{L}_n$ et $G \in \mathcal{L}_q \subset \mathcal{L}_n$.

On en déduit que $(F \wedge G) \in \mathcal{L}_{n+1} \subset \mathcal{L}_\infty$.

4. De même $(F \vee G) \in \mathcal{L}_\infty$ pour F et G dans \mathcal{L}_∞ .

Ainsi \mathcal{L}_∞ vérifie les propriétés 1. à 4.

Soit L vérifiant les propriétés 1. à 4.

On montre par récurrence sur n que alors $\mathcal{L}_n \subset L$ pour tout n .

↪ $\mathcal{L}_0 = \mathcal{A} \subset L$.

↪ Si $\mathcal{L}_n \subset L$ alors

pour tout $F \in \mathcal{L}_n$, on a $F \in L$ donc $\neg F \in L : \{\neg F, F \in \mathcal{L}_n\} \subset L$,

pour tout $F, G \in \mathcal{L}_n \subset L$ on a $(F \wedge G) \in E \in L : \{F \wedge G, F, G \in \mathcal{L}_n\} \subset L$,

de même $\{F \vee G, F, G \in \mathcal{L}_n\} \subset L$.

Ainsi l'union des ensembles, \mathcal{L}_{n+1} , est incluse dans L .

↪ La récurrence montre bien que $\mathcal{L}_n \subset L$ pour tout n .

On en déduit que $\mathcal{L}_\infty = \bigcup_{n \in \mathbb{N}} \mathcal{L}_n \subset L$ pour tout $L \in \mathbb{L}$.

\mathcal{L}_∞ vérifie les 4 propriétés et est inclus dans tout ensemble de \mathbb{L} ,

c'est le plus petit ensemble vérifiant les propriétés 1. à 4. d'où $\mathcal{L}_\infty = \mathcal{L}$.

Cette définition permet d'introduire les démonstrations par **induction structurelle**.

Un propriété est vraie pour toute formule si :

- ↪ elle est vraie pour toutes les variables propositionnelles,
- ↪ lorsqu'elle est vraie pour F , elle est vraie pour $\neg F$,
- ↪ lorsqu'elle est vraie pour F et G , elle est vraie pour $(F \wedge G)$,
- ↪ lorsqu'elle est vraie pour F et G , elle est vraie pour $(F \vee G)$.

De même on peut définir une fonction f sur \mathcal{L} par induction structurelle :

- ↪ on définit $f(x)$ pour toutes les variables propositionnelles,
- ↪ on définit $f(\neg F)$ à partir de $f(F)$,
- ↪ on définit $f(F \wedge G)$ à partir de $f(F)$ et $f(G)$,
- ↪ on définit $f(F \vee G)$ à partir de $f(F)$ et $f(G)$.

2-3 Simplification

L'écriture d'une formule contient beaucoup de parenthèses ce qui peut nuire à la lisibilité.

$$((\neg(x \wedge (y \vee (\neg z)))) \vee ((\neg y) \wedge x))$$

On décide alors d'un ordre de priorité sur les différents connecteurs :

- ↪ \neg est prioritaire devant \wedge
- ↪ \wedge est prioritaire devant \vee .
- ↪ dans une association de connecteurs (\wedge ou \vee) les priorités vont de la gauche vers la droite.

Exemples

- ↪ $((x \wedge y) \vee z)$ pourra s'écrire $x \wedge y \vee z$,
- ↪ $((\neg x) \wedge y)$ pourra s'écrire $\neg x \wedge y$

- ↪ $((((x_1 \vee x_2) \vee x_3) \vee x_4) \vee x_5)$ pourra s'écrire $x_1 \vee x_2 \vee x_3 \vee x_4 \vee x_5$.
- ↪ Par contre les parenthèses dans la formule $\neg(x \vee y)$ ne peuvent pas être enlevées sous peine de changer la nature de la formule.

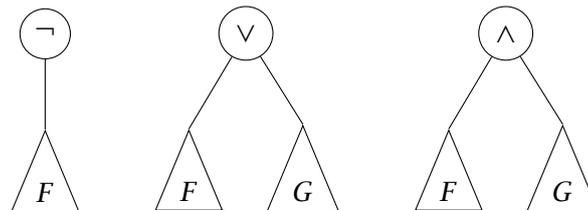
On admettra que la simplification est injective : deux formules distinctes ne peuvent pas donner la même simplification.

2-4 Arbre syntactique

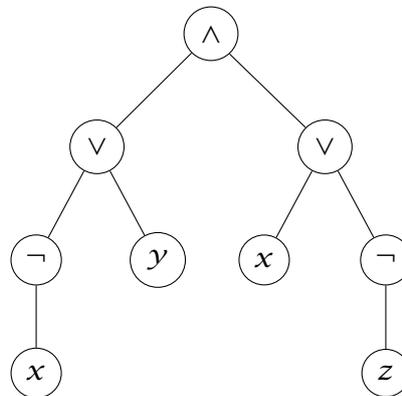
On définit par induction structurale un arbre à partir d'une formule.

- ↪ Une variable propositionnelle x est représentée par (x)

- ↪ Si F est représentée par $\triangle F$ et G par $\triangle G$ on représente $(\neg F)$, $(F \vee G)$ et $(F \wedge G)$ respectivement par



Ainsi la formule (simplifiée) $(\neg x \vee y) \wedge (x \vee \neg z)$ est représentée par :



Cette représentation en arbre donne un type possible pour les formules. On y représente les variables proportionnelles par des entiers

```
type formule = Variable of int
             | Non of formule
             | Ou of formule * formule
             | Et of formule * formule;;
```

3 Sémantique

La sémantique d'une formule est le caractère vrai (**V**) ou faux (**F**) de la formule.

Pour cela on définit une valeur de vérité à chacune des variables propositionnelles puis on calcule la valeur de vérité de la formule.

Valuation

Une valuation des variables propositionnelles est une application de \mathcal{A} , l'ensemble des variables propositionnelles, vers $\mathcal{B} = \{\mathbf{V}, \mathbf{F}\}$.

La sémantique consiste à prolonger une valuation v en une application \tilde{v} de l'ensemble des formules, \mathcal{L} , vers l'ensemble des valeurs de vérité, \mathcal{B} . Cela revient à donner une signification aux connecteurs.

On donne le sens suivant :

- ↪ la négation pour \neg ,
- ↪ ou (disjonction) pour \vee , c'est un ou non exclusif,
- ↪ et (conjonction) pour \wedge .

On définit \tilde{v} par induction structurelle :

- ↪ si $x \in \mathcal{A}$ alors $\tilde{v}(x) = v(x)$,
- ↪ si $F, G \in \mathcal{L}$ alors $\tilde{v}(\neg F)$, $\tilde{v}(F \wedge G)$ et $\tilde{v}(F \vee G)$ sont définis par les tables de vérité.

$\tilde{v}(F)$	$\tilde{v}(G)$	$\tilde{v}(\neg F)$	$\tilde{v}(F \wedge G)$	$\tilde{v}(F \vee G)$
V	V	F	V	V
V	F	F	F	V
F	V	V	F	V
F	F	V	F	F

Formules satisfaites

v désigne une valuation, F désigne une formule.

- ↪ Si $\tilde{v}(F)$ est **V** on dit que v **satisfait** F et que F **est satisfaite** par v .
- ↪ F est **satisfiable** s'il existe au moins une valuation qui satisfait F .
- ↪ F est une **tautologie** si elle est satisfaite par toutes les valuations.
- ↪ F est une **contradiction** si elle n'est satisfaite par aucune valuation.

Exemples : $F \vee \neg F$ est une tautologie et $F \wedge \neg F$ est une contradiction pour toute formule F .

Remarques

- ↪ F est une contradiction si et seulement si $\neg F$ est une tautologie.
- ↪ F est satisfiable si et seulement si elle n'est pas une contradiction.
- ↪ Si la formule F contient n variables propositionnelles distinctes, il existe 2^n valuations possibles de ces n variables. Pour tester si la formule est une tautologie (ou une contradiction) il faut évaluer la formule pour chacune de ces valuation.
- Déterminer si une formule est satisfiable se fait avec une complexité exponentielle.
- ↪ Par contre vérifier qu'une valuation donnée satisfait F ne prend que le temps de l'évaluation.

L'étude d'une formule peut se faire en calculant sa table de vérité : on représente toutes les valuations concernant les variables de la formule puis on détermine les valuation des composants de F pour aboutir à F . On indiquera souvent F à la place de $\tilde{v}(F)$.

Exemple : $F = (\neg x \vee y) \wedge (x \vee \neg z)$

v			\tilde{v}				
x	y	z	$\neg x$	$\neg x \vee y$	$\neg z$	$x \vee \neg z$	F
V	V	V	F	V	F	V	V
V	V	F	F	V	V	V	V
V	F	V	F	F	F	V	F
V	F	F	F	F	V	V	F
F	V	V	V	V	F	F	F
F	V	F	V	V	V	V	V
F	F	V	V	V	F	F	F
F	F	F	V	V	V	V	V

4 Équivalence

4-1 Définition

Formules équivalentes

Deux formules F, G sont équivalentes si, et seulement si, pour toute valuation ν , on a $\tilde{\nu}(F) = \tilde{\nu}(G)$.
 Cette équivalence est notée $F \equiv G$.

Deux formules équivalentes ont donc la même table de vérité.

Exemple : $G = x \wedge y \vee \neg(x \vee z)$

x	y	z	$x \wedge y$	$x \vee z$	$\neg(x \vee z)$	G
V	V	V	V	V	F	V
V	V	F	V	V	F	V
V	F	V	F	V	F	F
V	F	F	F	V	F	F
F	V	V	F	V	F	F
F	V	F	F	F	V	V
F	F	V	F	V	F	F
F	F	F	F	F	V	V

On retrouve la table de vérité de $F = (\neg x \vee y) \wedge (x \vee \neg z)$ donc $F \equiv G$.

Hérédité

Si F est équivalente à F' alors on peut remplacer une apparition de F par F' dans une formule G pour obtenir une formule équivalente G' .

Cela se démontre par induction structurale sur la construction de G à partir de F .

- ↪ Le cas de base est évident : $F \equiv F'$
- ↪ Si $G \equiv G'$ alors $\neg G \equiv \neg G'$
- ↪ Si $G \equiv G'$ et $H \equiv H'$ alors $(G \vee H) \equiv (G' \vee H')$ et $(G \wedge H) \equiv (G' \wedge H')$

Les implications ci-dessus sont des conséquences du calcul de \tilde{v} : $\tilde{v}(\neg G)$, $\tilde{v}(G \vee H)$ et $\tilde{v}(G \wedge H)$ ne dépendent que de $\tilde{v}(G)$ et $\tilde{v}(H)$ qui sont égaux respectivement à $\tilde{v}(G')$ et à $\tilde{v}(H')$.

Simplifications

- ↪ $\neg(\neg F) \equiv F$
- ↪ Si G est une tautologie alors $F \wedge G \equiv F$
- ↪ Si G est une contradiction alors $F \vee G \equiv F$

Distributivité

- ↪ $F \wedge (G \vee H) \equiv F \wedge G \vee F \wedge H$
- ↪ $F \vee G \wedge H \equiv (F \vee G) \wedge (F \vee H)$

Lois de Morgan

- ↪ $\neg(F \wedge G) \equiv \neg F \vee \neg G$
- ↪ $\neg(F \vee G) \equiv \neg F \wedge \neg G$

4-2 Formes normales

Dans l'étude des formules il peut être utile de travailler avec une forme standardisée des formules. Nous allons définir 2 telles formes. Bien qu'elles soient semblables elles correspondent en fait à deux états différents de la connaissance de la formule : la forme normale conjonctive représente une formule initiale, on ne sait pratiquement rien de sa table de vérité, la forme normale disjonctive est très proche de la table de vérité.

Formes normales

- Un **littéral** est une variable propositionnelle ou la négation d'une variable propositionnelle.
- Une **clause** (disjonctive) est une disjonction de littéraux.
C'est une formule de la forme $p_1 \vee \dots \vee p_n$ où $n \geq 1$ et p_1, \dots, p_n sont des littéraux.
- Une **clause conjonctive** est une conjonction de littéraux.
C'est une formule de la forme $p_1 \wedge \dots \wedge p_n$ où $n \geq 1$ et p_1, \dots, p_n sont des littéraux.
- Une **forme normale conjonctive** est conjonction de clause.
C'est une formule de la forme $C_1 \wedge \dots \wedge C_m$ où $m \geq 1$ et C_1, \dots, C_m sont des clauses.
- Une **forme normale disjonctive** est disjonction de clauses conjonctives : $D_1 \vee \dots \vee D_m$.

Par exemple la formule $F = x \wedge \neg y \vee y \wedge \neg z$ est une forme normale disjonctive.

On peut construire une forme normale disjonctive équivalente à F à partir de sa table de vérité.

↪ On considère chaque ligne qui donne un résultat **V**.

Elle est associée à une valuation v .

↪ À chaque variable x on associe le littéral x si $v(x) = \mathbf{V}$ ou $\neg x$ si $v(x) = \mathbf{F}$.

↪ On associe alors la formule D_v conjonction de ces littéraux

Forme normale canonique

La disjonction des formules D_v est une forme normale disjonctive équivalente à F

Les lois de Morgan impliquent que la négation d'une forme normale disjonctive est une forme normale conjonctive. On obtient donc une forme normale conjonctive équivalente à F en déterminant comme ci-dessus une forme normale disjonctive équivalente à $\neg F$ puis en calculant la négation. Exemple : on revient à $F = (\neg x \vee y) \wedge (x \vee \neg z)$

x	y	z	$\neg x$	$\neg x \vee y$	$\neg z$	$x \vee \neg z$	F	$\neg F$
V	V	V	F	V	F	V	V	F
V	V	F	F	V	V	V	V	F
V	F	V	F	F	F	V	F	V
V	F	F	F	F	V	V	F	V
F	V	V	V	V	F	F	F	V
F	V	F	V	V	V	V	V	F
F	F	V	V	V	F	F	F	V
F	F	F	V	V	V	V	V	F

Une forme normale disjonctive équivalente à F est :

$$(x \wedge y \wedge z) \vee (x \wedge y \wedge \neg z) \vee (\neg x \wedge y \wedge \neg z) \vee (\neg x \wedge \neg y \wedge \neg z)$$

Une forme normale disjonctive équivalente à $\neg F$ est :

$$(x \wedge \neg y \wedge z) \vee (x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge y \wedge z) \vee (\neg x \wedge \neg y \wedge z)$$

Ainsi F est équivalente à la forme normale conjonctive

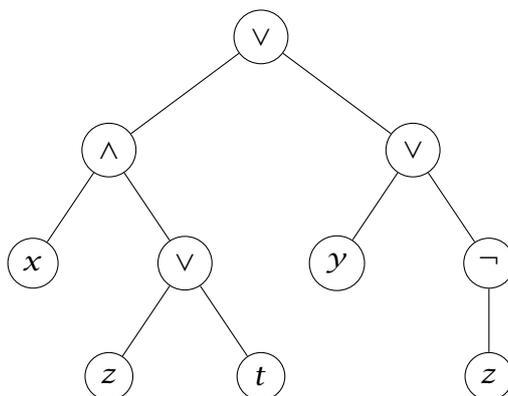
$$(\neg x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z)$$

On retrouve la forme normale conjonctive initiale en regroupant les clauses deux par deux et en utilisant la distributivité, la forme intermédiaire étant $(\neg x \vee y \vee z \wedge \neg z) \wedge (x \vee y \wedge \neg y \vee \neg z)$.

5 Exercices

Ex. 1 Dessiner la représentation arborescente de la formule $((x \vee (x \vee y)) \wedge (\neg x))$.

Ex. 2 Donnez la formule représentée par l'arbre suivant :



Ex. 3 Soit $x, y \in \mathcal{A}$ et soit v une valuation telle que $v(x) = \mathbf{V}$ et $v(y) = \mathbf{F}$, calculer $\tilde{v}(y \vee (x \wedge y))$

Ex. 4 Prouver les lois de simplification

$$\hookrightarrow \neg(\neg F) \equiv F$$

$$\hookrightarrow \text{Si } G \text{ est une tautologie alors } F \wedge G \equiv F$$

$$\hookrightarrow \text{Si } G \text{ est une contradiction alors } F \vee G \equiv F$$

Ex. 5 Prouver les lois de Morgan

$$\hookrightarrow \neg(F \wedge G) \equiv \neg F \vee \neg G$$

$$\hookrightarrow \neg(F \vee G) \equiv \neg F \wedge \neg G$$

Ex. 6 Prouver les lois de distributivité

$$\hookrightarrow F \wedge (G \vee H) \equiv F \wedge G \vee F \wedge H$$

$$\hookrightarrow F \vee G \wedge H \equiv (F \vee G) \wedge (F \vee H)$$

Ex. 7 Prouver les équivalences suivantes.

↪ **Idempotence** : $F \wedge F \equiv F$.

↪ **Absorption** : $F \wedge (F \vee G) \equiv F$.

Ex. 8 F est une formule de variables x_1, x_2, \dots, x_n .

V_F est l'ensemble des valuations v des n variables telles que $v(F) = \mathbf{V}$.

Pour une valuation v et pour chaque variable x on définit le littéral

$l_v(x) = x$ si $v(x) = \mathbf{V}$ et $l_v(x) = \neg x$ si $v(x) = \mathbf{F}$.

On pose alors $D_v = \bigwedge_{i=1}^n l_v(x_i)$ et $F_N = \bigvee_{v \in V_F} D_v$.

Prouver que la forme normale disjonctive F_N est équivalente à F .

Ex. 9 Tables de vérités avec 2 variables Écrire toutes les tables de vérité correspondant aux expressions construites sur un ensemble de deux variables propositionnelles. Identifier pour chaque table une expression simple dont elle est l'évaluation.

Ex. 10 If Then Else

On définit une fonction logique **ite** à 3 variables telle que **ite**(F, G, H) est équivalente à $(F \wedge G) \vee (\neg F \wedge H)$.

1. Quelle est la valeur de $\tilde{v}(\mathbf{ite}(F, G, H))$ si $\tilde{v}(F) = \mathbf{V}$?

Même question si $\tilde{v}(F) = \mathbf{F}$.

2. Exprimer $F \wedge G$ et $F \vee G$ par une formule simple comportant un seul **ite** .

3. On définit les constantes **0** et **1** telles que $\tilde{v}(\mathbf{0}) = \mathbf{F}$ et $\tilde{v}(\mathbf{1}) = \mathbf{V}$ pour toute valuation v .

Exprimer $\neg F$ par une formule comportant un seul **ite** et les constantes **0** et **1** .

Ex. 11 Implication

On définit le connecteur logique \Rightarrow par $F \Rightarrow G$ est équivalent à $\neg F \vee G$.

1. Prouver les équivalences (classiques) suivantes.

Contraposition : $[F \Rightarrow G] \equiv [\neg G \Rightarrow \neg F]$

Exportation : $[F \Rightarrow (G \Rightarrow H)] \equiv [F \wedge G \Rightarrow H]$

2. Prouver la loi de Pierce : $((F \Rightarrow G) \Rightarrow F) \Rightarrow F$ est une tautologie.
3. Soit p, q et r trois variables propositionnelles, déterminer les tables de vérité des formules $F = (p \Rightarrow q) \vee (q \Rightarrow r)$ et $G = (p \vee q) \Rightarrow (q \wedge r)$.

Ex. 12 Si et seulement si

On définit le connecteur logique \Leftrightarrow par

$[F \Leftrightarrow G] \equiv [(\neg F \vee G) \wedge (\neg G \vee F)]$.

Prouver que $[F \Leftrightarrow G] \equiv [F \wedge G \vee \neg F \wedge \neg G]$.

Montrer que $F \equiv G$ si et seulement si $F \Leftrightarrow G$ est une tautologie.

Ex. 13 Le connecteur de Sheffer

Le connecteur de Sheffer, noté $|$, est défini par :

$$[F|G] \equiv [\neg F \vee \neg G]$$

1. Montrez que les formules $\neg F$ et $F|F$ sont équivalentes.
2. Montrez que $F \vee G$ et $(F|F)|(G|G)$ sont des formules équivalentes.
3. En déduire que toute formule est équivalent à une formule ne contenant que des variables propositionnelles et le connecteur de Sheffer.
4. On ajoute un nouveau type

```
type formules = VarS of int |Shf of formules * formules;;
```

Écrire une fonction qui transforme une formule avec le type défini dans le cours en une formule du nouveau type.

Ex. 14 Le désert Vous êtes perdu sur une piste dans le désert. Vous arrivez à une bifurcation. Chacune des deux pistes est gardée par un sphinx que vous pouvez interroger. Les pistes peuvent soit conduire à une oasis soit se perdre dans un désert profond (au mieux elles conduisent toutes à une oasis, au pire elles se perdent toutes les deux).

Vous disposez des informations suivantes :

- A : le sphinx de droite dit : "*Une au moins des deux pistes conduit à une oasis*".
- B : le sphinx de gauche dit : "*La piste de droite se perd dans le désert*".
- C : vous savez que les sphinx disent tous les deux la vérité ou bien mentent tous les deux.

D est la proposition "*Il y a une oasis au bout de la route de droite*" et G est la proposition "*Il y a une oasis au bout de la route de gauche*".

1. Exprimer par une formule les affirmations A et B .
2. Exprimer alors la connaissance C .
3. Résoudre l'énigme.

Ex. 15 Détecteur de pannes Une nouvelle série de composants informatiques dédiés au raisonnement logique a été conçue de manière à faciliter la détection de pannes. Chaque processeur effectue des raisonnements logiques et peut-être soit en état de fonctionnement normal, soit en état de panne. Il se comporte de la manière suivante :

- un processeur qui fonctionne ne peut affirmer que des propositions vraies,
- un processeur en panne ne peut affirmer que de propositions fausses.

Un ordinateur est composé de trois processeurs qui possèdent la même mémoire, donc les mêmes connaissances. Périodiquement, un ingénieur vient interroger l'ordinateur pour déterminer si certains processeurs sont en état de panne.

Lors d'une séance de test, l'ingénieur pose les deux questions suivantes au processeur 1.

- Est-ce que les processeurs 2 et 3 sont en état de fonctionnement normal ?
Le processeur répond oui.
- Est-ce que le processeur 2 est en état de fonctionnement normal ?
Le processeur répond non.

Déterminer l'état de chaque processeur.

Ex. 16 Kjalts et lyops Dans un futur lointain, l'espèce humaine a découvert une autre espèce consciente. L'étude de cette espèce a permis de découvrir qu'elle est capable de percevoir si quelqu'un dit la vérité ou un mensonge. Les membres de cette espèce respectent les règles de politesse suivantes lors des discussions au sein d'un groupe.

→ Les orateurs doivent rester constants au cours d'une discussion : soit ils disent toujours la vérité, soit ils mentent toujours.

→ Si un orateur dit la vérité alors l'orateur suivant doit également dire la vérité.

→ Si le sujet de la discussion change, les orateurs sont libres de changer leurs comportements.

1. Vous assistez à une discussion sur les moyens d'attaque et de défense que peut posséder la faune de cette planète entre trois membres de cette espèce, appelés *A*, *B* et *C*.

A dit "Le *kjalt* peut avoir un dard ou des griffes.

B dit "Non, il n'a pas de dard.

C dit "Il a des pinces et des griffes.

Déterminer le (ou les) moyen(s) d'attaque et de défense que peut posséder un *kjalt*.

2. *C* quitte le groupe. La discussion change de sujet pour parler de la flore de la planète.

A dit "Un *lyop* peut être de couleur mauve mais pas de couleur jaune.

B dit "Il ne peut pas être de couleur verte.

A dit "Il ne peut être de couleur verte que s'il peut être de couleur jaune.

Déterminer la (ou les) couleur(s) possible(s) pour un *lyop*.

Chapitre IV

Graphes

1	Vocabulaire	56
1-1	Graphes non orientés	58
1-2	Connexité	59
2	Implémentations	61
2-1	Type abstrait	61
2-2	Liste des arêtes	61
2-3	Matrice d'adjacence	62
2-4	Tableau d'adjacence	63
2-5	Complexité	65
3	Parcours	66
3-1	Parcours récursif	66
3-2	Parcours en profondeur	68
3-3	Parcours général	70
3-4	Parcours en largeur	72
4	Applications des parcours	74
4-1	Composantes connexes	74
4-2	Plus court chemin	75
5	Exercices	77

Résumé

Les graphes sont la formalisation de l'assemblage d'entités avec relations : on gère donc, en plus de l'ensemble des objets, l'ensemble des relations qui peuvent lier deux objets. Dans ce chapitre nous nous restreindrons au cas où le choix est binaire : il y a relation ou il n'y a pas relation. Dans le chapitre suivant nous généraliserons au cas où une relation peut avoir une étiquette (ce sera le plus souvent un nombre).

1 Vocabulaire

Graphe

- Un graphe orienté est un couple (S, A) où
- ↪ S est un ensemble fini de **sommets** ou **noeuds**,
 - ↪ A est une partie de $S \times S \setminus \{(x, x) ; x \in S\}$, les **arêtes** ou arcs.
 - ↪ Si $(a, b) \in A$, a est l'**origine** de l'arc et b est sa **terminaison**.

On suppose donc que les arêtes ne sont pas des boucles : $a \neq b$.

On nommera aussi V l'ensemble des sommets (VERTEX/VERTICES ou NODES en anglais) et E l'ensemble des arêtes (EDGES en anglais).

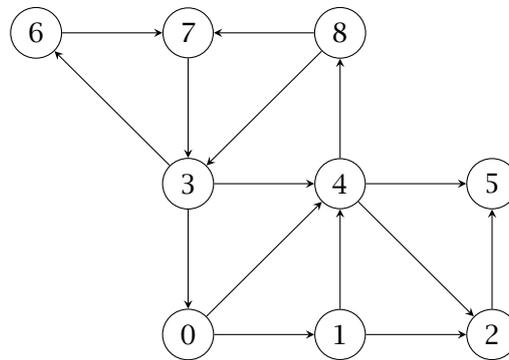
Exemple : $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$

$$A = \{(0, 1), (0, 4), (1, 2), (1, 4), (2, 5),$$

$$(3, 0), (3, 4), (3, 6), (3, 8), (4, 2),$$

$$(4, 5), (4, 8), (6, 7), (7, 3), (8, 7)\}$$

Un graphe sera représenté par un symbole pour chaque sommet et une flèche pour chaque arête.



Le plus souvent l'ensemble des sommets sera $\{0, 1, 2, \dots, n - 1\}$.

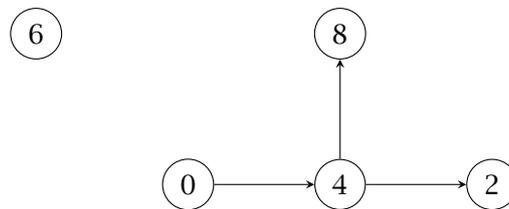
Le cardinal de S , $|S|$, est la taille du graphe.

$|A|$ donne une idée de la densité du graphe, c'est un entier compris entre 0 et $|S|(|S| - 1)$.

Graphe induit

Si $G = (S, A)$ est un graphe et si S' est une partie de S , le graphe induit par S' est $G' = (S', A \cap S' \times S')$.

Par exemple pour $S' = \{0, 2, 4, 6, 8\}$ le graphe induit par S' est



Degrés

s est un sommet d'un graphe G .

↪ Le **degré sortant** de s , $d^+(s)$, est le nombre d'arcs d'origine s .

↪ Le **degré rentrant** de s , $d^-(s)$, est le nombre d'arcs vers s .

↪ Les **voisins** de s sont les sommets s' tels que la paire (s, s') soit un arc. Le nombre de voisins est égal au degré sortant.

1-1 Graphes non orientés

Un cas particulier est celui où les arêtes forment un ensemble symétrique : dès qu'il y a un arc entre a et b alors il existe un arc entre b et a .

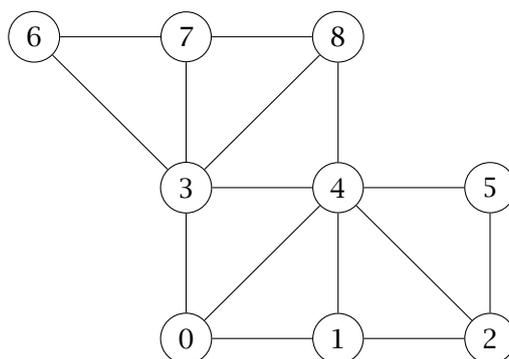
On simplifiera la donnée du graphe en considérant que les arêtes ne sont pas des couples mais des parties à deux éléments¹ de S .

Graphe non orienté

Un **graphe** non orienté est un couple (S, A) où

- ↪ S est un ensemble fini,
- ↪ A est une partie de $\mathcal{P}_2(S)$ ensemble des parties à deux éléments de S .

Il sera représenté avec des arêtes non fléchées.



Degré

s est un sommet d'un graphe non orienté G .

- ↪ Le **degré** de s , $d(s)$, est le nombre d'arcs dont s est un sommet.
- ↪ Les **voisins** de s sont les sommets s' tels que $\{s, s'\}$ soit un arc. Le nombre de voisins de s est égal au degré $d(s)$.

¹ Il ne peut pas y avoir de boucle de s vers lui-même.

1-2 Connexité

Chemins

↪ Un **chemin** de a vers b est une suite finie d'arcs de la forme

$$(a, a_1), (a_1, a_2), (a_2, a_3), \dots, (a_{k-1}, b)$$

On le notera aussi $a \rightarrow a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_{k-1} \rightarrow b$.

↪ La **longueur** d'un chemin est le nombre d'arcs qui le composent.

↪ Un **circuit** est un chemin d'un sommet vers lui-même.

↪ Un chemin de a vers b est **élémentaire** si tous les sommets sont distincts.

↪ Un **cycle** est un circuit dont tous les sommets sont distincts à l'exception du sommet initial et du sommet final.

Dans le cas d'un graphe non orienté la notion de cycle doit être modifiée.

En effet, pour toute arête (u, v) on a le cycle (u, v, u) qui est élémentaire au sens d'un graphe orienté.

Cycle

Un cycle d'un graphe non orienté est un chemin

$$((a_0, a_1), (a_1, a_2), \dots, (a_{n-2}, a_{n-1}), (a_{n-1}, a_0))$$

avec $n \geq 3$ et $a_i \neq a_j$ pour i et j distincts.

Graphe (fortement) connexe

Un graphe **orienté** est fortement connexe si, pour tous sommets a et b il existe un chemin entre a et b .

Dans le cas d'un graphe non orienté on dit que le graphe est connexe.

Le graphe orienté donné en exemple ci-dessus n'est pas fortement connexe car il n'existe aucun chemin entre 5 et un autre sommet. Par contre l'exemple non orienté est connexe.

Sommets équivalents

2 sommets a et b d'un graphe orienté sont **équivalents** s'il existe un chemin entre a et b et un chemin entre b et a .
C'est une relation d'équivalence.

Composantes (fortement) connexes

Les composantes fortement connexes d'un graphe orienté sont les classes d'équivalence pour cette relation.
Dans le cas d'un graphe non orienté on parle de composantes connexes.

Les composantes fortement connexes de l'exemple sont $\{0, 1, 3, 4, 6, 7, 8\}$, $\{2\}$ et $\{5\}$.

2 Implémentations

Nous allons définir les implémentations les plus courantes de la structure de graphe.

Le choix se fera en fonction des complexités des opérations importantes et parfois de l'occupation en mémoire.

Les graphes orientés ou non seront souvent représentés de la même façon : il faudra doubler les arêtes dans le cas d'un graphe non orienté.

2-1 Type abstrait

On note `graphe` le type de graphe.

On choisira des types dans lesquels l'ensemble des arêtes est mutable : on pourra ajouter ou enlever des arêtes sans créer un nouveau graphe.

Par contre l'ensemble des sommets sera fixé.

Les opérations de base dont on aura besoin pour utiliser la structure de graphe sont :

- ↪ la création d'un graphe sans arête de taille n , `creerGraphe : int -> graphe`,
- ↪ le calcul de la taille d'un graphe, `taille : graphe -> int`,
- ↪ le test d'existence d'une arête, `estArete : graphe -> int -> int -> bool`,
- ↪ l'ajout d'une arête, `ajouter : graphe -> int -> int -> unit`,
- ↪ le retrait d'une arête, `retirer : graphe -> int -> int -> unit`,
- ↪ la liste des voisins d'un sommet, `voisins graphe -> int -> int list`,
- ↪ la liste de toutes les arêtes, `aretes graphe -> (int*int) list`

2-2 Liste des arêtes

Une idée naturelle est de représenter un graphe par l'ensemble des arêtes. On lui ajoutera le nombre de sommets.

```
type graphe = {taille:int ; mutable aretes:(int*int) list};;
```

La traduction des fonctions est classique : voir l'exercice 7

Cette implémentation est peu utilisée car presque toutes les opérations ont une complexité de l'ordre de $|A|$.

2-3 Matrice d'adjacence

Une représentation plus efficace consiste à maintenir toutes les arêtes possibles sous la forme d'une matrice de taille $n \times n$ pour représenter un graphe de taille n . Pour un graphe non valué la valeur de la matrice aux indice i et j sera un booléen dont la valeur VRAI signifie que (i, j) est une arête. Le graphe non orienté défini ci-dessus donne la matrice suivante où on a écrit 0 pour FAUX et 1 pour VRAI.

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

Une telle représentation donne un accès rapide aux arêtes de sommets donnés au détriment d'un encombrement en mémoire de taille n^2 . Elle est à privilégier quand le graphe est dense.

Les premières fonctions sont immédiates

```
let creerGraphe n = Array.make_matrix n n false;;
```

```
let taille g = Array.length g;;
```

```
let estArete g s1 s2 = g.(s1).(s2);;
```

```
let ajouter g s1 s2 = g.(s1).(s2) <- true;;
```

```
let retirer g s1 s2 = g.(s1).(s2) <- false;;
```

La recherche des voisins ou des arêtes crée une liste pas-à-pas

```

let voisins g s =
  let v = ref [] in
  for i = 0 to (taille g - 1) do
    if g.(s).(i)
    then v := i::(!v) done;
  !v;;

```

Code IV.1 Voisins d'un sommet d'un graphe défini par une matrice d'adjacence

```

let aretes g =
  let a = ref [] in
  let n = taille g in
  for i = 0 to (n - 1) do
    for j = 0 to (n - 1) do
      if g.(i).(j)
      then a := (i,j)::(!a) done done;
  !a;;

```

Code IV.2 Arêtes d'un graphe défini par une matrice d'adjacence

2-4 Tableau d'adjacence

La représentation précédente permet un accès direct aux arêtes.

On peut souhaiter accéder directement aux voisins, en particulier dans la cas d'un graphe peu dense où le parcours de la ligne de la matrice est pénalisant.

Pour cela on peut implémenter un graphe sous la forme d'un tableau (indexé par les sommets)² de listes de voisins.

Le graphe initial (non orienté) donne le tableau

```

[[[1; 4]; [2, 4]; [5]; [0; 4; 6]; [2; 5; 8]; []; [7]; [3]; [3; 7]]]

```

Ici encore, il y a des fonctions immédiates.

```

let creerGraphe n = Array.make n [];;

```

² On peut remplacer le tableau par un dictionnaire si les sommets ne sont pas indicés par des entiers.

```
let taille = Array.length;;
```

```
let voisins g s = g.(s);;
```

La gestion individuelle des arêtes demande de parcourir les listes.
En effet on n'ajoute pas une arête si elle existe déjà.

```
let estArete g s1 s2 =
  let rec app a liste =
    match liste with
    | []          -> false
    | t::q when t = a -> true
    | t::q       -> app a q in
  app s2 g.(s1);;

let ajouter g s1 s2 =
  if not estArete g s1 s2
  then g.(s1) <- s2::(g.(s1));;
```

Code IV.3 Ajout d'une arête dans un graphe défini par un tableau d'adjacence

```
let enlever g s1 s2 =
  let rec oter s liste =
    match liste with
    | []          -> []
    | x::q when x = s -> q
    | x::q       -> x::(oter s q) in
  g.(s1) <- oter s2 (g.(s1));;
```

Code IV.4 Élimination d'une arête dans un graphe défini par un tableau d'adjacence

La liste des arêtes mélange accès à un tableau et traitement d'une liste.

```

let aretes g =
  let n = taille g in
  let ar = ref [] in
  let rec ajoute0 s liste =
    match liste with
    | [] -> []
    | t::q -> (s, t)::(ajoute0 s q) in
  for i = 0 to (n -1) do
    ar := (ajoute0 i g.(i))@(!ar) done;
  !ar;;

```

Code IV.5 Liste des arêtes dans un graphe défini par un tableau d'adjacence

Ici encore il faudra dédoubler les appels à ajouter et à retirer si le graphe est non orienté.

2-5 Complexité

- Dans le cas de la liste des arêtes les opérations ont une complexité qui ne dépend que du nombre des arêtes $m = |A|$.
- Dans le cas d'une implémentation par matrice d'adjacence les opérations ont une complexité qui ne dépend que de la taille du graphe, $n = |S|$.
- Dans le cas d'une implémentation par tableau d'adjacence les opérations dépendent de la taille des liste : on note d le maximum des degrés (sortant) du graphe.
Si on ne veut utiliser que les tailles on remarque que $d \leq n = |S|$.

fonction	liste des arêtes	matrice d'adjacence	tableau d'adjacence
estArete	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(d) = \mathcal{O}(n)$
ajouter	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(d) = \mathcal{O}(n)$
retirer	$\mathcal{O}(m)$	$\mathcal{O}(1)$	$\mathcal{O}(d) = \mathcal{O}(n)$
voisins	$\mathcal{O}(m)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
aretes	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$	$\mathcal{O}(m)$

Dans la suite nous utiliserons surtout la fonction `voisins` : on calculera les complexités en supposant que les graphes sont implémentés sous forme de tableaux d'adjacence.

3 Parcours

Le parcours d'un graphe consiste à visiter les sommet d'un graphe à partir d'un sommet d'origine en suivant les voisins des sommets visités. Nous allons voir plusieurs méthodes de parcours qui se différencient selon la mémorisation que l'on fait des différents sommets que l'on doit visiter.

Contrairement aux arbres, les graphes peuvent contenir des cycles : il est donc indispensable de garder trace des sommets déjà visités afin de ne pas les re-visiter, on évite ainsi de parcourir des cycles indéfiniment.

Quand les sommets sont les entiers de 0 à $n - 1$ on peut utiliser pour cela un tableau de booléens.

3-1 Parcours récursif

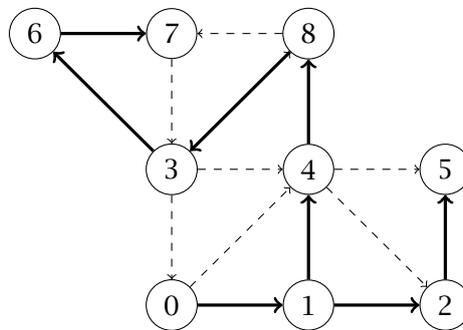
Le parcours récursif consiste à marquer un sommet comme vu puis visiter récursivement ses voisins non encore vus. On peut tester si le sommet a été vu avant ou après l'appel récursif. On choisit ici de le faire immédiatement lors du traitement du sommet.

```
let parcours g s0 =  
  let vus = Array.make (taille g) false in  
  let rec visiter s =  
    if not vus.(s)  
    then begin vus.(s) <- true;  
              List.iter visiter (voisins g s) end  
  in visiter s0;  
  vus;;
```

Code IV.6 Parcours récursif

On applique cet algorithme pour l'exemple donné ci-dessus ; on suppose que la fonction `voisins` renvoie la liste des sommets par ordre croissant.

sommet visité	tableau vu	sommet d'origine
0	[V, F, F, F, F, F, F, F, F]	∅
1	[V, V, F, F, F, F, F, F, F]	0
2	[V, V, V, F, F, F, F, F, F]	1
5	[V, V, V, F, F, V, F, F, F]	2
4	[V, V, V, F, V, V, F, F, F]	1
8	[V, V, V, F, V, V, F, F, V]	4
3	[V, V, V, V, V, V, F, F, V]	8
6	[V, V, V, V, V, V, V, F, V]	3
7	[V, V, V, V, V, V, V, V, V]	6



Terminaison et complexité

On n'appelle la fonction que pour les voisins d'un sommet non vu.

Il n'y a donc qu'un appel de la fonction par arête du graphe traversée : le nombre d'appels est fini donc l'algorithme termine.

De plus le nombre d'appel est majoré par le nombre d'arêtes.

La création du tableau est de complexité linéaire en sa taille, $|S|$.

Au total la complexité est un $\mathcal{O}(|A| + |S|)$.

Preuve

1. On commence par montrer que **tous les sommets vus sont accessibles** depuis s_0 ; on peut le faire par récurrence sur le nombre de sommets déjà marqués vus.

Le premier marqué comme vu est s_0 , trivialement accessible depuis s_0 .

On suppose que les p sommets marqués comme vus sont accessibles depuis s_0 et que l'on visite le sommet s , non encore vu. La fonction `visiter s` a été appelée lors du traitement d'un sommet s' qui a été marqué comme vu à ce moment.

D'après l'hypothèse de récurrence s' est accessible depuis s_0 .

s a alors été appelé en tant que voisin de s' donc le chemin de s_0 à s' peut se prolonger par l'arête (s', s) ; ainsi s est accessible depuis s_0 .

Les $p + 1$ sommets marqués comme vus sont donc accessibles.

On a ainsi prouvé le résultat souhaité.

2. S'il existait un sommet t accessible depuis s_0 mais non visité, on considère un chemin de s vers t : $s = s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_{k-1} \rightarrow s_k = t$.
 s_0 est marqué comme vu et t ne l'est pas : il existe donc $i \in \{1, 2, \dots, k\}$ tel que s_{i-1} est vu mais s_i n'est pas marqué vu. On a donc appelé s_{i-1} mais pas s_i . On aboutit à une contradiction car tous les voisins de s_{i-1} , en particulier s_i , ont été testés ; comme s_i n'est pas marqué comme vu il y aurait du être traité puis marqué comme vu.
 Ainsi **tous les sommets accessibles depuis s_0 sont marqués comme vus**.
3. L'ensemble des sommets marqués comme vus est bien l'ensemble des sommets accessibles depuis s_0 .

Remarque : la complexité calculée ci-dessus n'est valide que dans le cas où la fonction `voisins` est de complexité constante, c'est le cas de la représentation par liste d'adjacences.

3-2 Parcours en profondeur

Nous allons donner une version itérative du parcours. Une des raisons pour cela est de gérer la pile qui, dans le cas de la pile des appels récursifs peut être limitée (c'est le cas en Python). De plus cela donnera un schéma qui peut être généralisé.

```

on place s0 dans une pile
tant que la pile n'est pas vide
  on dépile un élément
  s'il n'a pas encore été vu
    on le marque comme vu
    on empile ses voisins
  
```

Pour obtenir le même ordre que dans le cas de la récursivité, on peut inverse l'ordre des sommets dans la liste des voisins.

Cependant on remarque que l'on peut placer plusieurs fois le même sommet dans la pile ; on diminue la complexité spatiale (l'espace occupé par la pile) en n'empilant que les sommets non encore visités. L'ordre de parcours des éléments sera alors différent.

On va utiliser les piles du module **Stack** de OCAML.

- ↪ `Stack.create` : `unit -> 'a t` crée une pile vide,
- ↪ `Stack.is_empty` : `'a t -> bool` teste si une pile est vide,
- ↪ `Stack.push` : `'a -> 'a t -> unit` ajoute un élément à une pile,
- ↪ `Stack.pop` : `'a t -> 'a` retire le dernier élément d'une pile et le renvoie,
- ↪ `Stack.top` : `'a t -> 'a` renvoie le dernier élément d'une pile **sans le retirer**.

```

let parcoursProfondeur g s0 =
  let vus = Array.make (taille g) false in
  let attente = Stack.create() in
  let traiter s = if not vus.(s)
                  then (Stack.push s attente;
                        vus.(s) <- true) in
  Stack.push s0 attente;
  vus.(s0) <- true;
  while not (Stack.is_empty attente) do
    let s = Stack.pop attente in
    List.iter traiter (voisins g s) done;
  vus;;

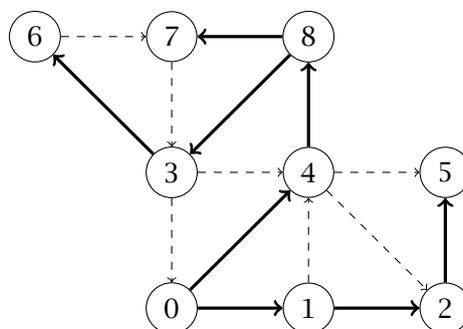
```

Code IV.7 Parcours en profondeur d'un graphe

Voici le parcours du graphe orienté donné en exemple. Les voisins sont renvoyés dans l'ordre décroissant, ils seront donc dans l'ordre croissant en sortie de pile.

sommet traité	pile	tableau vus
initialisation	0	[T, F, F, F, F, F, F, F, F]
0	1, 4	[T, T, F, F, T, F, F, F, F]
1	2, 4	[T, T, T, F, T, F, F, F, F]
2	5, 4	[T, T, T, F, T, T, F, F, F]
4	8	[T, T, T, F, T, T, F, F, T]
8	3, 7	[T, T, T, T, T, T, F, T, T]
3	6, 7	[T, T, T, T, T, T, T, T, T]
6	7	[T, T, T, T, T, T, T, T, T]
7		[T, T, T, T, T, T, T, T, T]

On indique ici les arêtes utilisées lors du passage à true des sommets



3-3 Parcours général

On peut généraliser la construction précédente.

```

on place s0 dans un ensemble
on le marque comme vu
tant que l'ensemble n'est pas vide
  on extrait un des éléments de cet ensemble
  on place ses voisins non vus dans l'ensemble
  en les marquant comme vus

```

On a vu deux types d'ensembles différents des piles : les files et les files de priorité. Dans le cas des files de priorité, il sera utile de reporter le marquage comme vu au moment de l'extraction, en effet la priorité peut évoluer en fonction de ce qui a été vu.

On peut remplacer le tableau de booléens par un tableau des sommets qui indiquera depuis quel sommet les sommets vus ont été marqués comme vus. Cela permettra de construire un chemin depuis s_0 lorsqu'un sommet est accessible. Pour marquer qu'un sommet n'a pas été vu on peut donner la valeur -1 comme valeur initiale ou on peut utiliser un type optionnel : `None` indiquera un sommet non vu, `Some k` indiquera que le sommet a été vu depuis k .

```

let parcoursGeneral g s0 =
  let pere = Array.make (taille g) None in
  let attente = ensembleVide () in
  ajouter s0 attente;
  pere.(s0) <- Some s0;
  while not (estVide ensemble) do
    let t = extraction ensemble in
    let traiter s = if pere.(s) = None
                    then begin pere.(t) <- Some s;
                              ajouter s ensemble end in
    List.iter traiter (voisins g s) done;
  pere;;

```

À partir du tableau des pères, on calcule le chemin de s_0 vers une destination d .

```

let chemin g s0 d =
  let pere = parcoursGeneral g s0 in
  let rec aux s chemin =
    match peres.(s) with
    |None -> [] (* pas de chemin *)
    |Some p when p = s0 -> s0::chemin
    |Some p -> aux p (p::chemin)
  in aux d [d];;

```

Code IV.8 Calcul du chemin depuis s_0 **Terminaison**

On ne place un sommet s dans l'ensemble que si s n'a pas encore été traité donc on ne lit qu'une fois les voisins de chaque sommet. On ne va itérer la boucle qu'au plus $|A|$ fois.

Preuve

On prouve que les sommets qui ont un père sont exactement ceux qui sont accessibles depuis s_0 de la même manière que dans le cas récursif.

Complexité

On a créé un tableau de taille $|S|$ qui demande une complexité proportionnelle à $|S|$. On appelle la fonction traiter au plus $|A|$ fois et on y effectue au plus une adjonction dans l'ensemble. Si $i(n)$ et $e(n)$ sont les complexités d'insertion et d'extraction d'un élément dans un ensemble de taille n et si $v(n)$ est la complexité du calcul de la liste des voisins dans un graphe de taille n alors la complexité du parcours est majorée par $|A|. (i(|A|) + e(|A|)) + |S|.v(|S|)$.

Dans le cas d'opérations à coûts constant on retrouve une complexité en $\mathcal{O}(|A| + |S|)$.

3-4 Parcours en largeur

Si on utilise une file on effectue un parcours en largeur.

On peut utiliser module **Queue** de OCAML.

- ↪ `Queue.create` : `unit -> 'a t` crée une file vide,
- ↪ `Queue.is_empty` `'a t -> bool` teste si une file est vide,
- ↪ `Queue.add` `'a -> 'a t -> unit` ajoute un élément à une file,
- ↪ `Queue.take` `'a t -> 'a` retire le premier élément d'une file et le renvoie,
- ↪ `Queue.top` `'a t -> 'a` renvoie le premier élément d'une file **sans le retirer**.

```

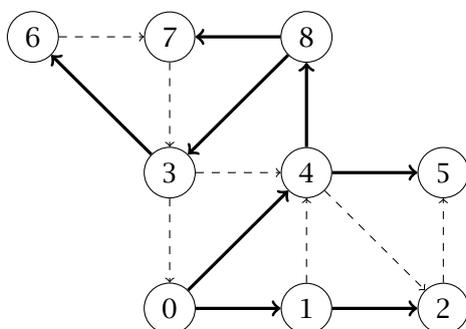
let parcoursLargeur g s0 =
  let pere = Array.make (taille g) None in
  let attente = Queue.create() in
  Queue.add s0 attente;
  pere.(s0) <- Some s0;
  while not (Queue.is_empty attente) do
    let s = Queue.take attente in
    let traiter t = if pere.(t) = None
                    then begin pere.(t) <- Some s;
                               Queue.add s attente end in
    List.iter traiter (voisins g s) done;
  pere;;

```

Code IV.9 Parcours en largeur d'un graphe

Voici le parcours du graphe orienté donné en exemple. Les voisins sont renvoyés dans l'ordre croissant. Pour permettre une meilleure clarté, on a utilisé -1 à la place de None et *k* à la place de Some *k*.

sommet traité	file	tableau pere
initialisation	0	[0, -1, -1, -1, -1, -1, -1, -1, -1]
0	1, 4	[0, 0, -1, -1, 0, -1, -1, -1, -1]
1	4, 2	[0, 0, 1, -1, 0, -1, -1, -1, -1]
4	2, 5, 8	[0, 0, 1, -1, 0, 4, -1, -1, 4]
2	5, 8	[0, 0, 1, -1, 0, 4, -1, -1, 4]
5	8	[0, 0, 1, -1, 0, 4, -1, -1, 4]
8	3, 7	[0, 0, 1, 8, 0, 4, -1, 8, 4]
3	7, 6	[0, 0, 1, 8, 0, 4, 3, 8, 4]
7	6	[0, 0, 1, 8, 0, 4, 3, 8, 4]
6		[0, 0, 1, 8, 0, 4, 3, 8, 4]



On peut implémenter une file avec des complexités constantes pour l'ajout et le retrait d'un élément donc, dans le cas d'un graphe défini par un tableau d'adjacences, la complexité du parcours est en $\mathcal{O}(|A|)$.

4 Applications des parcours

4-1 Composantes connexes

Comme un parcours permet de déterminer tous les sommets accessibles depuis un sommet donné, il permet de calculer, **dans le cas d'un graphe non orienté**, la composante connexe contenant ce sommet.

Si on part de chaque sommet on pourra ainsi déterminer les composantes connexes d'un graphe non orienté.

- On maintient un tableau des numéros des composantes dont les éléments sont des entiers. Un sommet a été vu si et seulement si la valeur correspondante dans le tableau n'est pas -1 .
- On balaie les sommets. Quand un sommet n'a pas été vu, on incrémente le numéro de composante connexe et on l'attribue à tous les sommets accessibles.

```
let composantesConnexes g =
  let n = taille g in
  let composante = Array.make n (-1) in
  let numero = ref (-1) in
  let rec visiter k s =
    if composante.(s) = -1
    then (composante.(s) <- k;
          List.iter (visiter k) (voisins g s)) in
  for i = 0 to (n-1) do
    if composante.(i) = -1
    then (numero := 1 + !numero;
          visiter !numero i) done;
  composante;;
```

Code IV.10 Calcul des composantes connexes

Dans les exercices cet algorithme sera aussi invoqué sous le nom de parcours récursif complet.

4-2 Plus court chemin

On peut souhaiter connaître le plus court chemin, lorsqu'il en existe, entre deux sommets. On démontre qu'on obtient un chemin minimum en nombre d'arêtes avec le parcours en largeur.

Plus court chemin

Le chemin renvoyé par le parcours en largeur a une longueur minimale parmi les chemins de s_0 à s dans le graphe G (pour s accessible depuis s_0).

On note $\delta(a, b)$ la longueur du plus court chemin de a à b dans G .

Démonstration

1. On montre l'invariant de boucle suivant.

Si la file contient les éléments (a_1, a_2, \dots, a_p) (dans l'ordre de leur insertion) alors il existe un entier d tel que

$$1. d = \delta(s_0, a_1) \leq \delta(s_0, a_2) \leq \dots \leq \delta(s_0, a_p) \leq d + 1,$$

2. tous les sommets s tels que $\delta(s_0, s) \leq d$ ont déjà été insérés dans la file³.

Au départ la file ne contient que s_0 avec $d_0 = 0 = \delta(s_0, s_0)$

et s_0 est le seul sommet à distance 0 : l'invariant est vérifié.

On suppose que l'invariant est vérifié avant les instructions de la boucle `while`.

On retire le premier élément de la liste, a_1 , sa distance à s_0 est d .

On ajoute ses voisins non encore visités. Soit s un de ces sommets (s'il en existe).

On a $d = \delta(s_0, a_1)$ donc il existe un chemin de s_0 à a_1 de longueur d .

L'arête (a_1, s) permet de prolonger ce chemin en un chemin de s_0 à s de longueur $d + 1$; on en déduit que $\delta(s_0, s) \leq d + 1$.

D'après l'hypothèse de récurrence, tous les sommets de distance d ou moins ont déjà été insérés donc on a $\delta(s_0, s) \geq d + 1$.

Ainsi $\delta(s_0, s) = d + 1$ pour tous les sommets ajoutés, notés a_{p+1}, \dots, a_q .

Deux cas sont possibles

↪ Soit on a $p \geq 2$ et $\delta(s_0, a_2) = \delta(s_0, a_1) = d$.

La suite devient $(a_2, \dots, a_p, a_{p+1}, \dots, a_q)$ qui vérifie la propriété 1 avec le même entier d .

Comme d est inchangé, la propriété 2 est maintenue.

↪ Soit on a $p = 1$ ou $p \geq 2$ et $d_2 = p + 1$.

³ Certains peuvent en être sortis

Dans ce cas la file peut être vide ou ne contenir que des sommets dont la distance à s_0 est $d + 1$ et la propriété 1 est vérifiée avec l'entier $d + 1$.

D'après l'hypothèse de récurrence, tous les sommets à distance d ou moins ont déjà été insérés et comme il n'en reste plus dans la file, ils ont été traités.

Pour tout sommet à distance $d + 1$ de s_0 , il existe un chemin

$$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_{d-1} \rightarrow s_d \rightarrow s_{d+1} = s$$

s_d est alors à distance d de s_0 donc a été traité.

On considère le premier sommet traité parmi ceux (à distance d de s_0) qui ont s comme voisin. Lors du traitement de ce sommet, s est placé dans la file.

Ainsi tous les sommets à distance $d + 1$ de s_0 sont dans la file.

La propriété 2 est vérifiée.

L'invariant est donc conservé à l'étape suivante.

2. Comme l'invariant est toujours vérifié, la démonstration a montré que chaque sommet à distance $d + 1$ de s_0 a été inséré dans la file lors du traitement d'un sommet à distance $d + 1$ de s_0 . C'est ce sommet qui est placé dans le tableau des antécédents.

Pour tout sommet accessible, s , la fonction `chemin` renvoie un chemin

$$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \dots \rightarrow s_{d-1} \rightarrow s_d = s$$

On sait que $d(s_0, s_k) = d(s_0, s_{k-1}) + 1$ et $d(s_0, s_0) = 0$ donc $d(s_0, s_p) = p$: le chemin renvoyé est bien de longueur minimale.

5 Exercices

Ex. 1 Nombre d'arêtes

Si $G = (S, A)$ est un graphe orienté montrer que $\sum_{s \in S} d^+(s) = \sum_{s \in S} d^-(s) = |A|$.

Que se passe-t-il dans le cas d'un graphe non orienté ?

Ex. 2 Degrés égaux

Prouver que, dans un graphe non orienté, il existe deux sommets ayant le même degré.

Ex. 3 S' est une composante fortement connexe d'un graphe (S, A) .

Prouver le graphe induit par S' est fortement connexe.

Ex. 4 Caractérisation des composantes connexes s_0 est un sommet d'un graphe non orienté $G = (S, A)$.

1. Prouver que la composante connexe contenant s_0 est l'ensemble des sommets accessibles depuis s_0 c'est-à-dire l'ensemble des sommets s tels qu'il existe un chemin entre s_0 et s .
2. Prouver que la composante connexe contenant s_0 est la plus grande partie $S' \subset S$ telle que le graphe induit par S' est connexe.

Ex. 5 Minoration du nombre d'arêtes Prouver que si G est connexe alors $|A| \geq |S| - 1$.

Ex. 6 Isthme

Un isthme d'un graphe connexe non orienté est une arête dont la suppression déconnecte le graphe.

Soit $e = \{x, y\}$ une arête d'un graphe connexe non orienté G .

Montrer l'équivalence des trois propriétés suivantes.

1. (x, y) est un isthme de G .
2. (x, y) est le seul chemin simple de G reliant x et y .
3. (x, y) n'est inclus dans aucun cycle simple de G .

Ex. 7 Fonctions pour une liste d'arêtes Écrire les fonctions associées au type abstrait dans le cas d'une représentation par la liste des arêtes.

Ex. 8 Traductions Écrire une fonction qui traduit un graphe défini par la liste des arêtes en sa représentation par une matrice d'adjacence et la fonction réciproque.

Ex. 9 Écrire les fonctions qui traduisent un graphe défini par une matrice d'adjacence en sa représentation par un tableau d'adjacence et réciproquement.

Ex. 10 Écrire les fonctions qui traduisent un graphe défini par un tableau d'adjacence en sa représentation par une liste d'arêtes et réciproquement.

Ex. 11 Écrire une fonction qui calcule les composantes connexes sous la forme d'une liste de listes, ces dernières regroupant les éléments de chaque composante connexe. Il est demandé de faire le calcul directement, pas de convertir le résultat du chapitre.

Arbres

Dans cette partie on considère des graphes **non orientés**.

Graphe acyclique

Un graphe non orienté est acyclique s'il ne contient pas de cycle simple.

Ex. 12 Nombre d'arêtes

Prouver que si $G = (S, A)$ est acyclique alors $|A| \leq |S| - 1$.

Prouver que si $G = (S, A)$ est connexe alors $|A| \geq |S| - 1$.

Ex. 13 Graphe circulaire

Un graphe non orienté où tous les sommets sont d'ordre 2 est-il nécessairement un cycle ?

À quelle condition est-ce un cycle ?

Arbre

Un arbre est un graphe non orienté connexe et acyclique.

Ex. 14 Caractérisation des arbres

Prouver que $G = (S, A)$ est un arbre si et seulement si l'une des conditions suivantes est réalisée.

1. G est connexe et $|A| = |S| - 1$.
2. G est acyclique et $|A| = |S| - 1$.
3. G est connexe et n'est plus connexe si on retire une arête quelconque (toutes ses arêtes sont des isthmes).
4. G est acyclique et devient cyclique si on ajoute une arête.

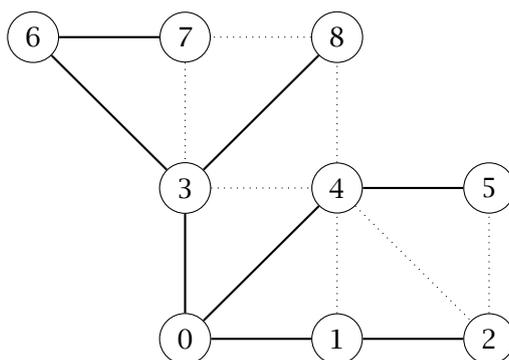
Ex. 15 Bouts d'un arbre

Prouver que si un graphe est un arbre alors il admet au moins deux sommets de degré 1. Que se passe-t-il s'il en a exactement 2 ?

Arbre couvrant

Un arbre est un arbre couvrant d'un graphe si admet les mêmes sommets et si les arêtes de l'arbre sont des arêtes du graphe.

Un arbre couvrant possible.



On remarque que, dans les exemples du cours, les arêtes joignant les sommets à leurs voisins non vus, forment un arbre couvrant. C'est le cas général.

Ex. 16 Parcours et arbre couvrant

Prouver que, lors d'un parcours, les arêtes joignant un sommet visité s à un voisin non encore visité forment un arbre couvrant de la composante connexe contenant s_0 .

Ex. 17 Calculs d'arbres couvrants

Modifier les parcours pour qu'ils renvoient un arbre couvrant sous la forme d'une liste d'arêtes.

Algorithme de Warshall

Ex. 18 Distances depuis un sommet

Modifier le parcours en largeur afin qu'il renvoie un tableau des distances depuis le sommet origine. Les sommets inaccessibles pourront avoir -1 pour distance.

L'algorithme précédent ne calcule les distances que depuis un sommet.

On peut souhaiter calculer les distances entre chaque paire de sommets.

On peut proposer un autre algorithme en utilisant la programmation dynamique : on calcule les longueurs des chemins entre i et j en incluant un par un les sommets intermédiaires possibles.

On note $d_k(i, j)$ la distance minimale entre i et j en ne passant que par les sommets $0, 1, \dots, k-1$ (en dehors de l'origine et de l'extrémité).

On a $d_0(i, i) = 0$, $d_0(i, j) = 1$ si (i, j) est une arête et $d_0(i, j)$ est infinie sinon.

Si n est la taille du graphe alors un chemin minimum, s'il existe, est de longueur au plus $n-1$. On peut choisir donc n comme longueur infinie.

Pour aller de i à k en ne passant que par les sommets $0, 1, \dots, k-1$ et k on peut

↪ ne passer que par les sommets $0, 1, \dots, k-1$

↪ passer par k , dans ce cas le chemin se coupe en un chemin de i à k et un chemin de k à j , chacun de ces deux chemins ne passant que par les sommets $0, 1, \dots, k-1$.

La relation de récurrence est donc $d_{k+1}(i, j) = \min(d_k(i, j), d_k(i, k) + d_k(k, j))$.

Ex. 19 Algorithme de Warshall

Programmer cet algorithme. Quelle est la complexité ?

Ex. 20 Algorithme de Warshall amélioré

Modifier le programme précédent pour qu'il renvoie une fonction `chemin` telle que `chemin i j` retourne un chemin de longueur minimale entre i et j .

On pourra maintenir une matrice dont les éléments ont un type

```
type passage = Sans | Direct | Milieu of int;;
```

qui signifie l'absence de chemin, un chemin direct ou un chemin passant par k , `Milieu k` étant inséré lorsque $d_{k+1}(i, j) = d_k(i, k) + d_k(k, j)$.

Graphes orienté sans circuits

Ex. 21 Lemme

Prouver que, dans un graphe orienté sans circuit, il existe un sommet de degré sortant nul.

Ordre topologique

Un ordre topologique d'un graphe de taille n est une bijection σ de l'ensemble des sommets vers $\{0, 1, \dots, n - 1\}$ telle que, pour toute arête (i, j) on a $\sigma(i) < \sigma(j)$.

Ex. 22 Caractérisation

Prouver qu'un graphe orienté est sans circuit si et seulement si il admet un ordre topologique.

On propose l'algorithme suivant qui renvoie la liste des sommets d'un graphe.

```

let tri_final g =
  let n = taille g in
  let vus = Array.make n false in
  let tri = ref [] in
  let rec visiter s =
    if not vus.(s)
    then begin vus.(s) <- true;
              List.iter visiter (voisins g s);
              tri = s :: !tri
            end in
  for i = 0 to (n-1) do visiter i done;
  !tri;;

```

Code IV.11 Ordre final du parcours récursif

Ex. 23 **Ordre induit par le parcours récursif**

Prouver que, dans un graphe orienté sans circuit, l'ordre des sommets renvoyé par la fonction `tri_final` est un ordre topologique.

Le parcours récursif complet permet aussi de tester si un graphe est acyclique.

Lors de ce parcours un sommet peut être dans trois états :

- ↪ il peut ne pas encore avoir été atteint,
- ↪ il peut être en cours de traitement,
- ↪ il peut avoir été visité.

On définit un type correspondant à ces états :

```
type etatSommet = NonVu | EnCours | Vu;;
```

On peut alors donner une modification du parcours complet.

```
let test g =
  let n = taille g in
  let etat = Array.make n NonVu in
  let rec visiter s =
    if etat.(s) = NonVu
    then begin etat.(s) <- EnCours;
              List.iter visiter (voisins g s);
              etat.(s) <- Vu
            end in
  for i = 0 to (n-1) do visiter i done;
  etat;;
```

La fonction devrait retourner un tableau rempli `Vu`.

Ex. 24 **Détection de cycles**

Prouver qu'un graphe orienté admet un circuit si et seulement si un sommet est dans un état `EnCours` lors d'un appel.

En déduire un algorithme détectant si un graphe est acyclique.

Composantes fortement connexes

On commence par généraliser les résultats de la partie précédente.

On emploie le calcul de l'ordre final de visite, `tri_final` : **code IV.11, page 82**.

Ex. 25 Ordre partiel

On suppose que, dans un graphe orienté, il existe un chemin entre s et t mais il n'existe pas de chemin entre t et s . Prouver que s est placé avant t dans le résultat de `tri_final`

Graphe transposé

Si $G = (S, A)$ est un graphe le graphe transposé, G^T , est le graphe (S, A^T) où A^T est l'ensemble des arêtes retournées :

$$A^T = \{(t, s) ; (s, t) \in A\}$$

Le graphe transposé est donc le graphe obtenu en inversant les flèches.

Un graphe est non-orienté si et seulement si il est égal à son transposé.

Ex. 26 Calcul du transposé

Écrire une fonction `transpose g` qui renvoie le transposé de g .

Ex. 27 Composantes du transposé

Prouver que G et G^T ont les mêmes composantes fortement connexes.

Ex. 28 Clôture des composantes

G est un graphe orienté, G^T est son transposé.

On parcourt G^T par un parcours récursif complet en lisant les sommets dans l'ordre de la liste `ordre` obtenue par `tri_final g..`

Prouver que le traitement d'un point ne traite que des sommets de sa composante fortement connexe.

Prouver alors le traitement du premier point d'une composante fortement connexe qui apparaît dans la liste `ordre` traite tous les points de la composante et eux seuls.

Ex. 29 Calcul des composantes fortement connexes

Écrire une fonction qui calcule les composantes connexes fortement connexes sous la forme d'une liste de listes, ces dernières regroupant les éléments de chaque composante connexe.

L'algorithme suggéré ici est celui de **Kosaraju**.

Chapitre

V

Graphes valués

1	Introduction	88
1-1	Définitions	88
1-2	Implémentations	89
2	Plus court chemin	91
2-1	Algorithme de Floyd-Warshall	91
2-2	Algorithme de Dijkstra	93
3	Exercices	95

Résumé

Les graphes que nous avons étudiés permettent de modéliser les relations entre entités.

Nous allons introduire la possibilité d'une valeur des relations : les arêtes peuvent avoir un poids. On peut alors prolonger ce poids aux chemins et introduire un questionnement sur l'optimisation de ces valeurs :

- ↪ quel sont les chemins de poids minimum ?
- ↪ comment recouvrir à moindre coût le graphe ?
- ↪ quel est le flux maximum entre deux points ?

1 Introduction

1-1 Définitions

Graphe valué

Un graphe valué sur \mathcal{B} est un triplet (S, A, w) où

- ↪ (S, A) est un graphe non orienté
- ↪ w est une application de A vers \mathcal{B} ;
 $w(i, j)$ est le **poids** de l'arête (i, j) .

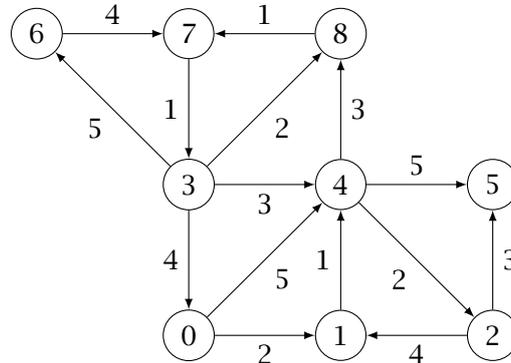
Les poids des arêtes seront souvent des entiers ou des réels.

Nous nous restreindrons à des poids strictement positifs.

Dans le cas d'un graphe non orienté on impose $w(a, b) = w(b, a)$.

Exemple : $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ et $A = \{(0, 1), (0, 4), (1, 4), (2, 1), (2, 5), (3, 0), (3, 4), (3, 6), (3, 8), (4, 2), (4, 5), (4, 8), (6, 7), (7, 3), (8, 7)\}$

On détermine le poids avec $w(i, j) = (i + j) \bmod 5 + 1$.



Poids d'un chemin

Le poids d'un chemin $c = ((a, a_1), (a_1, a_2), (a_2, a_3), \dots, (a_{k-1}, b))$ est la somme des poids des arêtes qui le composent,

$$W(c) = \sum_{i=1}^k w(a_{i-1}, a_i) \text{ avec } a_0 = a \text{ et } a_k = b.$$

Le chemin vide (direct) entre i et i a le poids 0.

1-2 Implémentations

Les types devront gérer les poids, dont le type sera symbolisé par 'a'. En particulier on introduit la fonction

```
poids : int -> int -> 'a
```

telle que `poids i j` renvoie $w(i, j)$.

Les fonctions `voisins` et `aretes` ajouteront le poids dans leurs retours et `ajouter` prendra le poids en paramètre.

Matrice d'adjacence

Une matrice d'adjacence permet de stocker facilement le poids.

Cependant il faut gérer l'absence d'arc. On marquera souvent la non-existence d'un arc par une arête de poids infini.

Cet infini pourra être représenté par la constante `max_int` dans le cas de poids entiers ou, plus simplement, à l'aide d'un type composé.

```

type 'a arete = Infini|Poids of 'a;;

type 'a graphe == 'a arc array array;;

```

L'exemple ci-dessus est implémenté par

$$\begin{pmatrix} \infty & 2 & \infty & \infty & 5 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 1 & \infty & \infty & \infty & \infty \\ \infty & 4 & \infty & \infty & \infty & 3 & \infty & \infty & \infty \\ 4 & \infty & \infty & \infty & 3 & \infty & 5 & \infty & 2 \\ \infty & \infty & 2 & \infty & \infty & 5 & \infty & \infty & 3 \\ \infty & \infty \\ \infty & 4 & \infty \\ \infty & \infty & \infty & 1 & \infty & \infty & \infty & \infty & \infty \\ \infty & 1 & \infty \end{pmatrix}$$

Listes d'adjacence

Pour l'implémentation sous forme de listes d'adjacence on peut maintenir la liste des voisins avec leur poids.

```

type 'a graphe == (int*'a) list array;;

```

L'exemple ci-dessus est implémenté par

```

[| [1, 2; 4, 5]; [4, 1]; [1, 4; 5, 3];
  [0, 4; 4, 3; 6, 5; 8, 3]; [2, 2; 5, 5; 8, 3]; [];
  [7, 4]; [3, 1]; [7, 1] |]

```

2 Plus court chemin

Dans cette partie nous allons chercher à répondre à la question

"*Quel est le poids minimum d'un chemin entre deux sommets a et b d'un graphe ?*".

On rappelle que les poids sont supposés strictement positifs.

Existence d'un chemin de poids minimum

S'il existe un chemin c entre a et b dans un graphe valué à poids positifs alors il existe un chemin de poids minimum entre a et b .

La démonstration est proposée dans l'exercice 3.

2-1 Algorithme de Floyd-Warshall

La première méthode que nous allons étudier détermine les poids minimum entre toute paire de sommet.

L'algorithme va utiliser l'idée de la programmation dynamique : on va calculer les poids minimum en autorisant les chemins de a vers b à passer par un ensemble de sommets de plus en plus grand.

↪ On note $\Gamma_k(i, j)$ l'ensemble des chemins simples (c'est-à-dire sans boucle) de i à j qui ne passent, entre i et j , que par des sommets appartenant à $\{0, 1, \dots, k-1\}$. En particulier $\Gamma_0(i, j)$, pour $i \neq j$, est vide si (i, j) n'est pas un arc et est restreint à (i, j) si $(i, j) \in A$.

Par convention $\Gamma_k(i, i)$ n'est pas vide, il est réduit au chemin vide de poids nul.

↪ Les chemins appartenant à $\Gamma_k(i, j)$ sont

soit les chemins de $\Gamma_{k-1}(i, j)$ s'ils ne passent pas par k

soit les concaténations d'un chemin de $\Gamma_{k-1}(i, k)$ et d'un chemin de $\Gamma_{k-1}(k, j)$ sinon.

On a donc $\Gamma_k(i, j) = \Gamma_{k-1}(i, j) \cup \Gamma_{k-1}(i, k) \cdot \Gamma_{k-1}(k, j)$.

↪ On note $p_k(i, j)$ le poids minimum des chemins de $\Gamma_k(i, j)$.

Ainsi $p_n(i, j)$ est le poids minimum des chemins de a vers b .

Le résultat sur les chemins donne le calcul

$$p_k(i, j) = \min\{p_{k-1}(i, j), p_{k-1}(i, k) + p_{k-1}(k, j)\}$$

On note P_k la matrice dont les coefficients sont les $p_k(i, j)$.

On peut alors implémenter le calcul de la matrice des poids minimums.

Le programme commence par initialiser P_0 à partir des voisins pondérés de chaque sommet puis effectue une boucle pour le calcul des P_k .

```

let poidsMin g =
  let n = taille g in
  let p = Array.make_matrix n n Infini in
  for i = 0 to (n-1) do
    List.iter (fun (j,k) -> p.(i).(j) <- Poids k)
              (voisins g i);
    p.(i).(i) <- Poids 0 done;
  for k = 0 to (n-1) do
    for i = 0 to (n-1) do
      for j = 0 to (n-1) do
        p.(i).(j) <- petit (plus p.(i).(k) p.(k).(j))
                          p.(i).(j)
      done
    done
  done;
  p;;

```

Code V.1 Algorithme de Floyd-Warshall

On modifie la matrice `p` "en place" : l'exercice 4 montre que cela est possible. Les fonctions `plus` et `petit` sont proposées dans l'exercice 6.

Complexité de l'algorithme de Floyd-Warshall

La complexité de `poidsMin` est en $\mathcal{O}(n^3)$.

Démonstration : l'initialisation a une complexité au pire en $\mathcal{O}(n^2)$ et les trois boucles imbriquées donnent la complexité.

2-2 Algorithme de Dijkstra

La deuxième méthode que nous allons étudier ne calcule que les chemins minimums depuis un sommet s_0 fixé.

On notera $d(s, s')$ le poids minimum d'un chemin de s à s' , on parlera de **distance** de s à s' . Ce n'est pas une distance au sens mathématique dans le cas d'un graphe orienté car on peut avoir $d(s, s') \neq d(s', s)$. Le principe est de calculer, à chaque étape, le sommet le plus proche de s_0 : c'est un algorithme **glouton** (*greedy* en anglais) qui progresse en cherchant un optimum local et qui permet à la fin d'obtenir un optimum global.

On commence par la recherche du sommet le plus proche de s_0 : cela doit être un des voisins de s_0 . En effet un sommet s qui n'est pas voisin de s_0 n'est accessible depuis s_0 qu'en passant par un voisin s , ainsi $d(s_0, s) \geq w(s_0, s)$.

Le plus proche sommet de s_0 est donc s_1 tel que $w(s_0, s_1) = \min\{w(s_0, s) ; s \text{ voisin de } s_0\}$.

On généralise ce raisonnement pour les sommets suivants.

→ On suppose déterminés les $p - 1$ sommets les plus proches de s_0 : s_1, s_2, \dots, s_{p-1} .

On cherche le suivant : s_p .

→ On note $S' = S \setminus \{s_0, s_1, \dots, s_{p-1}\}$: s_p doit être un sommet de S' tel que $d(s_0, s_p)$ soit minimale parmi les $d(s_0, s')$ pour $s' \in S'$.

→ Un chemin minimum de s_0 à s_p ne passe, en dehors de s_p , que dans $S \setminus S'$. En effet, s'il existait un point de S' dans ce chemin, celui-ci serait à une distance à s_0 inférieure à celle de s_p : ce qui est exclu.

→ Si l'avant-dernier sommet du chemin minimum de s_0 à s_p est s_k alors on peut écrire $d(s_0, s_p) = d(s_0, s_k) + w(s_k, s_p)$.

→ Ainsi s_p est un sommet de S' , connecté à un sommet de $S \setminus S'$ et qui réalise le minimum de $\{d(s_0, s) + w(s, s') ; s \in S' \text{ et } s' \text{ voisin de } s\}$.

La dernière assertion fournit un algorithme pour calculer pas-à-pas les distances.

Pour gérer les arêtes depuis les sommets visités on peut les entreposer dans une file de priorité qui renverra celle dont la distance possible est minimale.

On placera dans la file de priorité les triplets (s, t, d) où s est le père du sommet t , t est le sommet et d est le poids du chemin de s_0 à t qui commence par un chemin de poids minimum de s_0 à s et qui finit par l'arête (s, t) .

On notera qu'un sommet t peut être placé plusieurs fois dans la file d'attente.

La clé de comparaison pour la file est alors le troisième élément.

```
let cle (a, b, c) = c;;
```

Pour gérer les chemins on crée un tableau des pères, si on veut aussi la distance sans calcul on peut créer un tableau des distances.

On rappelle les fonctions d'une liste de priorité (type 'a fdp) :

```
creeVide : 'a -> 'a fdp
estVide : 'a fdp -> bool
ajouter : 'a -> 'a fdp -> unit()
premier : 'a fdp -> 'a
enlever : 'a fdp -> unit()
extraire : 'a fdp -> 'a
```

Dans le cas d'une file de priorité implémentée par un tas on peut ajouter un élément ou retirer le minimum en un temps en $\mathcal{O}(\log(n))$ où n est la taille du tas.

```
let fonctionCheminMin g s0 =
  let n = taille g in
  let fp = creeVide (0, 0, 0) in
  let pere = Array.make n (-1) in
  let distance = Array.make n (-1) in
  ajouter (s0, s0, 0) fp;
  while not (estVide fp) do
    let (s, t, d) = extraire fp in
    if distance.(t) = -1
    then begin
      distance.(t) <- d;
      pere.(t) <- s;
      List.iter (fun (u, w) -> ajouter (t, u, w+d) fp)
                (voisins g t)
    end done;
  let rec chemin j =
    if pere.(j) = -1
    then []
    else if pere.(j) = j
    then [j]
    else j :: (chemin pere.(j)) in
  fun i -> distance.(i), List.rev (chemin i);;
```

Code V.2 Algorithme de Dijkstra

Complexité de l'algorithme de Dijkstra

La complexité de l'algorithme est en $\mathcal{O}(p \log(p))$.

Démonstration : on parcourt un tas qui va recevoir toutes les arêtes si tous les sommets sont accessibles. Si p est le nombre d'arêtes la complexité est donc en $\mathcal{O}(p \log(p))$. Comme on a $p \leq n^2$ où n est le nombre de sommets on peut aussi majorer la complexité par un $\mathcal{O}(n^2 \log(n))$.

3 Exercices

Ex. 1 Implémentation des graphes valués avec des matrices d'adjacence Écrire les fonctions `creerGraphe`, `taille`, `existe`, `poids`, `ajouter`, `retirer`, `voisins` et `aretes` à l'aide de matrices d'adjacence.

Ex. 2 Implémentation des graphes valués avec des tableaux d'adjacence Écrire les fonctions `creerGraphe`, `taille`, `existe`, `poids`, `ajouter`, `retirer`, `voisins` et `arêtes` à l'aide de tableaux d'adjacence.

Ex. 3 Existence
 G est un graphe valué à poids positifs.
Montrer que s'il existe un chemin c entre a et b ($a \neq b$) alors il existe un chemin c' entre a et b sans boucle tel que $W(c') \leq W(c)$.
En déduire que, dans ce cas, il existe un chemin de poids minimum entre a et b .

Ex. 4 Calcul en place On reprend les notations de la partie 2-1.
Prouver que $p_{k-1}(i, k) = p_k(i, k)$ ainsi que $p_{k-1}(k, j) = p_k(k, j)$.
En déduire que l'on peut calculer les matrices P_k "en place" ; c'est-à-dire en utilisant une seule matrice et remplaçant $p(i, j)$ par $\min\{p(i, j), p(i, k) + p(k, j)\}$ pour tout les indices i et j à chaque étape.

Ex. 5 Pseudo-distance

On rappelle que d est le poids minimum d'un chemin de s à s' .

Prouver que d vérifie l'inégalité triangulaire : $d(s, s'') \leq d(s, s') + d(s', s'')$.

Ex. 6 Arithmétique avec l'infini

Écrire les fonctions `plus` et `petit` qui permettent de calculer la somme et le minimum pour deux éléments de type `'a` arc contenant un infini.

Ex. 7 Calcul du chemin

Modifier le programme de Floyd-Warshall pour qu'il renvoie une fonction `chemin` telle que `chemin i j` retourne un chemin de longueur minimale entre i et j .

On pourra maintenir une autre matrice dont les éléments ont un type

```
type passage = Sans | Direct | Milieu of int;;
```

qui signifie l'absence de chemin, un chemin direct ou un chemin passant par k , ce point étant inséré lors de la diminution de la distance au rang k .

Chapitre VI

Flots

1	Méthode de Ford et Fulkerson	98
1-1	Flot	98
1-2	Chemin améliorant	101
2	Coupure	104
3	Algorithme de Edmonds-Karp	106
3-1	Complexité	106
3-2	Implémentation	107

L'objet de ce complément est d'étudier un problème d'optimisation dans un graphe valué. Les notions vont s'appuyer sur de nombreuses définitions qui sont en dehors du programme. Les résultats sont proposés sous forme d'exercices.

1 Méthode de Ford et Fulkerson

Dans cette sous-partie nous allons définir et caractériser la notion de flot maximal. Cette caractérisation va permettre d'imaginer une méthode de calcul qu'il faudra préciser ensuite.

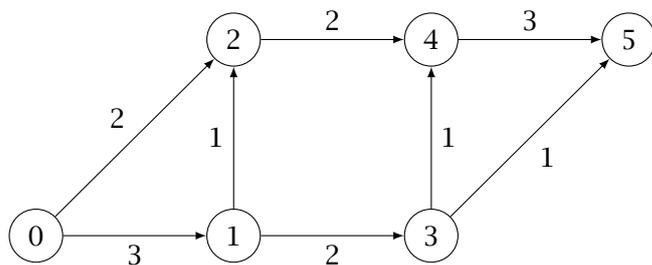
1-1 Flot

Réseau

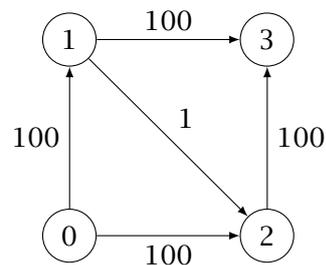
Un réseau est un quintuplé $R = (S, A, w, s, p)$ tel que

- ↪ $G = (S, A, w)$ graphe orienté valué,
- ↪ w est à valeurs dans $]0; +\infty[$,
- ↪ s est un sommet de G , la source,
- ↪ p est un sommet de G , le puits, avec $s \neq p$.

Exemples de réseaux



$R_1, s = 0, p = 5$



$R_2, s = 0, p = 3$

Capacité

La capacité du réseau est la fonction c de $S \times S$ vers \mathbb{R}^+ définie par

↪ $c(i, j) = w(i, j)$ si $(i, j) \in A$

↪ $c(i, j) = 0$ sinon.

Flot

Un flot sur un réseau $R = (S, A, w, s, p)$ est une fonction φ de $S \times S$ vers \mathbb{R} telle que

↪ $\varphi(i, j) \leq c(i, j)$ pour tous sommets i et j ,

↪ $\varphi(j, i) = -\varphi(i, j)$ pour tous sommets i et j

↪ $\sum_{j \in S} \varphi(i, j) = 0$ pour tout sommet i distinct de s et p .

La valeur du flot est $|\varphi| = \sum_{j \in S} \varphi(s, j)$.

Ex. 1 Nullité

Prouver que si φ est un flot d'un graphe et si ni (i, j) , ni (j, i) n'est une arête du graphe alors $\varphi(i, j) = 0$. En particulier $\varphi(i, i) = 0$.

Ex. 2 Valeur au puits

Prouver que, pour toute partie T de S , $\sum_{i \in T} \sum_{j \in T} \varphi(i, j) = 0$.

En déduire que $|\varphi| = \sum_{i \in S} \varphi(i, p)$.

Ex. 3 Flot associé à un chemin

On suppose que le graphe d'un réseau admet un chemin simple de s vers p :

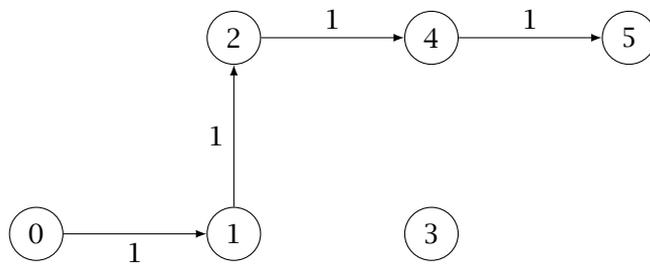
$$s = s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_{r-1} \rightarrow s_r = p$$

On définit $\omega = \min\{w(s_{i-1}, s_i) ; 1 \leq i \leq r\}$

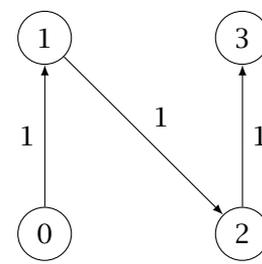
et φ par $\varphi(s_{i-1}, s_i) = \omega$, $\varphi(s_i, s_{i-1}) = -\omega$ et $\varphi(i, j) = 0$ pour les autres couples.

Prouver que φ est un flot (c'est le flot associé au chemin) avec $|\varphi| > 0$.

Exemples de flots associés à un chemin. On ne représente que les flots strictement positifs, les flots strictement négatifs correspondent aux inverses des arêtes.



$$|\varphi_1| = 1$$

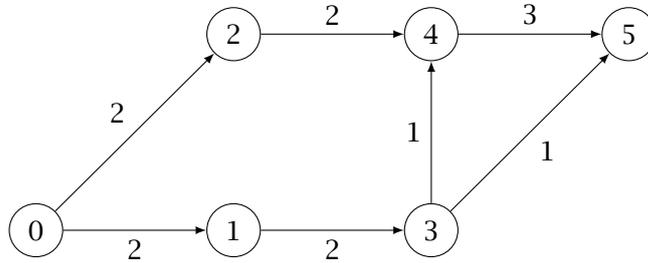


$$|\varphi_2| = 1$$

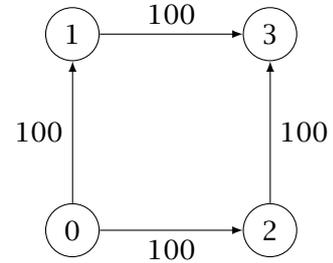
Propriétés

- Il existe au moins un flot : le flot nul défini par $\varphi(i, j) = 0$ pour tous sommets.
- Pour tout flot φ , on a $|\varphi| = \sum_{j \in S} \varphi(s, j) \leq \sum_{j \in S} c(s, j)$.
- L'ensemble des valeurs des flots est donc non vide et majoré, il admet une borne supérieure.
- On cherche un flot (s'il en existe) tel que $|\varphi|$ est maximum.

Exemples de flots maximaux



$|\varphi_{1,M}| = 43$



$|\varphi_{2,M}| = 200$

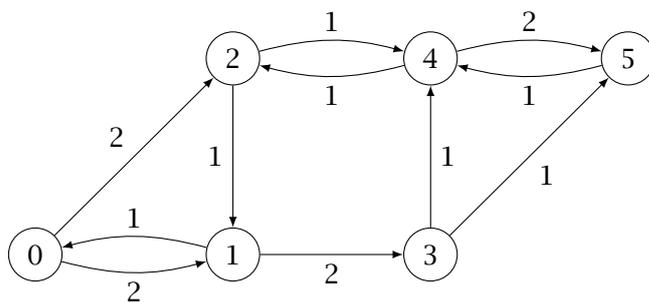
1-2 Chemin améliorant

Nous allons chercher à augmenter la valeur d'un flot.

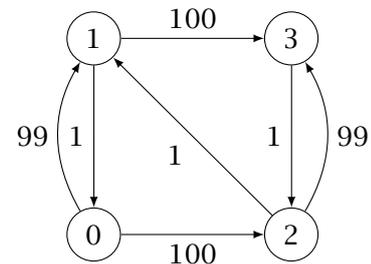
Réseau résiduel

Si φ est un flot sur le réseau $R = (S, A, w, s, p)$, le réseau résiduel associé est $R_\varphi = (S, A_\varphi, w_\varphi, s, p)$ où A_φ est l'ensemble des couples (i, j) tels que $\varphi(i, j) < c(i, j)$ et $w_\varphi(i, j) = c(i, j) - \varphi(i, j)$.

Exemples de réseaux résiduels : on part des flots associés à un chemin, vus à la page 100



R_{φ_1}



R_{φ_2}

On remarque qu'on a enlevé des arêtes de R et introduit des arêtes supplémentaires en sens inverse par rapport à celles de R . Ces arêtes inversées correspondent à la possibilité de diminuer le flot qui suit l'arête originale.

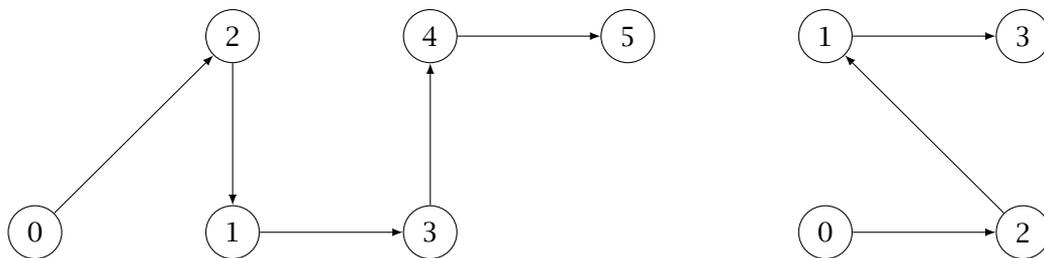
Ex. 4 Combinaison de flots

Prouver que si φ est un flot du réseau $R = (S, A, w, s, p)$ et si ψ est un flot du réseau R_φ alors $\varphi + \psi$ est un flot de R avec $|\varphi + \psi| = |\varphi| + |\psi|$.

Chemin améliorant

Si φ est un flot sur le réseau $R = (S, A, w, s, p)$, un chemin améliorant est un chemin de s vers p dans le réseau résiduel associé R_φ .

Exemples de chemins améliorants : on reprend les même exemples de flots

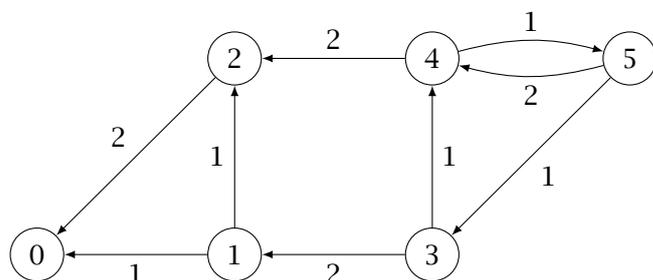


Ex. 5 Condition nécessaire

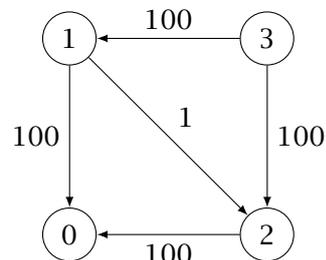
Prouver que si φ est un flot maximal de R alors il n'existe pas de chemin améliorant dans R_φ .

Exemples de réseaux résiduels pour des flots maximaux

Ils n'admettent pas de chemin de s à p .



$R_{\varphi_{1,M}}$



$R_{\varphi_{2,M}}$

La méthode de Ford-Fulkerson s'en déduit

- ↪ R est un réseau
- ↪ φ est le flot nul
- ↪ $R' \leftarrow R$, c'est le réseau résiduel associé à φ)
- ↪ **tant que** R' admet un chemin entre s et p
- ↪ on choisit un chemin de s à p
- ↪ on détermine le flot de R' associé à ce chemin
- ↪ $\varphi \leftarrow \varphi + \psi$
- ↪ $R' \leftarrow R_{\varphi}$
- ↪ φ est un flot maximal.

Cependant on n'a pas prouvé que l'absence de chemin améliorant caractérisait la maximalité du flot. Il faut prouver la réciproque du résultat de l'exercice 5.

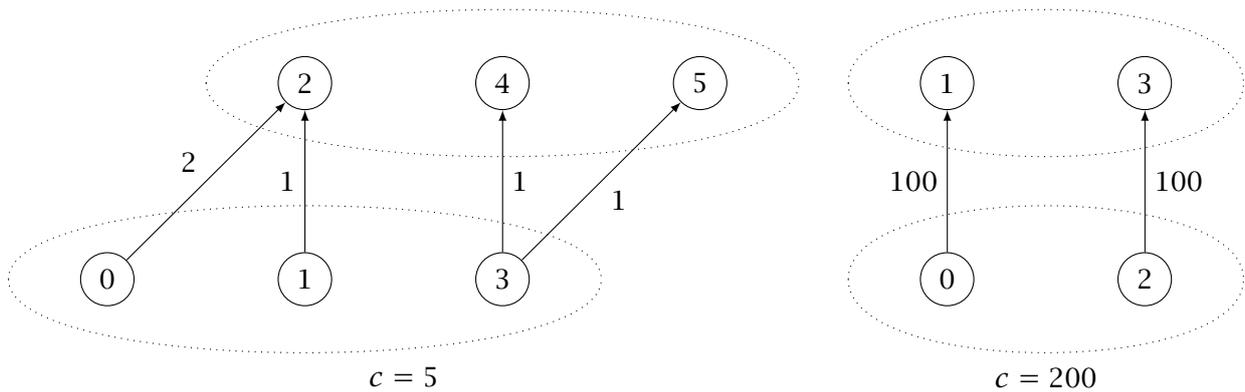
2 Coupure

On introduit une nouvelle notion qui va servir d'étape intermédiaire pour prouver l'algorithme de Fulkerson

Coupure

Une coupure d'un réseau $R = (S, A, w, s, p)$ est une partition (S_s, S_p) de l'ensemble S des sommets telle que $s \in S_s$ et $p \in S_p$.
 La capacité d'une coupure est $c(S_s, S_p) = \sum_{i \in S_s} \sum_{j \in S_p} c(i, j)$.

Exemples de coupures



Ex. 6 Majoration de la valeur d'un flot

Prouver que si φ est un flot du réseau $R = (S, A, w, s, p)$ et si (S_s, S_p) est coupure du réseau alors $|\varphi| \leq c(S_s, S_p)$.
 On pourra prouver qu'on a $|\varphi| = \sum_{i \in S_s} \sum_{j \in S_p} \varphi(i, j)$.

On retrouve un autre majorant pour les valeurs d'un flot : c'est la capacité d'une coupure.

Ex. 7 Coupure minimale pour un flot

Prouver que s'il existe une coupure (S_s, S_p) de R telle que $c(S_s, S_p) = |\varphi|$ pour un flot φ alors ce flot est maximal.

Une telle coupure sera dite **minimale** pour un flot φ .

Ex. 8 Existence d'une coupure minimale pour un flot maximal

Prouver que s'il n'existe pas de chemin améliorant dans R_φ alors il existe une coupure minimale pour φ dans R .

En rassemblant les exercices 7, 8 et 5 on a le

Théorème de Ford et Fulkerson

Si φ est un flot d'un réseau $R = (S, A, w, s, p)$ alors les propriétés suivantes sont équivalentes.

1. φ est un flot maximal.
2. Il existe une coupure minimale pour φ dans R .
3. Il n'existe pas de chemin améliorant dans R_φ .

Remarque Dans le cas de poids réels, le théorème ne permet pas d'obtenir un algorithme de calcul du flot maximal. En effet la suite strictement croissante des valeurs des flots n'atteint pas toujours sa limite après un nombre fini d'itérations selon le choix du chemin améliorant. La méthode n'est pas assurée de terminer.

Le cas de poids entiers est moins difficile.

Ex. 9 Complexité

Prouver que si les poids sont entiers la méthode de Ford-Fulkerson termine après au plus C itérations avec $C = \sum_{i \in S} c(s, i)$.

En considérant le graphe R_2 avec un choix systématique d'un chemin améliorant qui passe par la diagonale $(1, 2)$ montrer que ce majorant peut être atteint.

3 Algorithme de Edmonds-Karp

L'algorithme de Edmonds-Karp consiste à implémenter la méthode de Ford-Fulkerson en choisissant comme chemin améliorant celui que fournit un parcours en largeur depuis s .

Notations : rappels et définitions

- ↪ On part d'un réseau $R = (S, A, w, s, p)$.
- ↪ φ est un flot sur R , $R_\varphi = (S, A_\varphi, w_\varphi, s, p)$ est le réseau résiduel associé.
- ↪ S_φ est l'ensemble des sommets accessibles depuis s dans R_φ . Pour $t \in S_\varphi$, on note $d_\varphi(t)$ la longueur du plus court chemin (en nombre d'arêtes) de s à t dans R_φ .
- ↪ Si on a $p \in S_\varphi$, le parcours en largeur depuis s dans R_φ donne un chemin augmentant noté c .
- ↪ ψ est le flot de R_φ associé ; $\psi(a) = \omega$ pour toute arête de c où ω est poids minimal des arêtes de c (dans $R_{\varphi'}$).
- ↪ On note $\varphi' = \varphi + \psi$ le flot augmenté et $R_{\varphi'} = (S, A_{\varphi'}, w_{\varphi'}, s, p)$ est le réseau associé.
- ↪ Une arête de R_φ est dite **critique** si elle n'appartient pas à $R_{\varphi'}$.
- ↪ Une arête de $R_{\varphi'}$ est dite **émergente** si elle n'appartient pas à R_φ .

3-1 Complexité

Ex. 10 Caractérisations des arêtes critiques

Prouver que les arêtes critiques sont les arêtes de c de poids ω dans R_φ .

Ex. 11 Caractérisations des arêtes émergentes

Prouver que les arêtes émergentes sont les arêtes (i, j) n'appartenant pas à $A_{\varphi'}$ telles que (j, i) est une arête de c .

En déduire que i ne peut être égal à s , ni j égal à p .

Ex. 12 Décroissance de S_φ Prouver qu'on a $S_{\varphi'} \subset S_\varphi$.

Ex. 13 Augmentation des distances

On suppose que (i, j) est une arête émergente appartenant à un chemin minimal depuis s . Montrer qu'on a $d_{\varphi'}(j) - d_\varphi(j) = d_{\varphi'}(i) - d_\varphi(i) + 2$.

En déduire que $d_\varphi(t) \leq d_{\varphi'}(t)$ pour tout $t \in S_{\varphi'}$.

Ex. 14 Dénombrements

Montrer qu'une même arête ne peut être critique que $|S|/2$ fois au maximum.

En déduire l'algorithme de Edmonds-Karp ne peut faire plus de $|A| \times |S|$ itérations de recherche de chemin minimal.

Que peut-on en déduire pour la complexité de l'algorithme de Edmonds-Karp ?

3-2 Implémentation

On choisit de coder les graphes par leur matrice des poids, l'absence d'arête est représentée par un poids nul.

La fonction de flot est aussi représentée par la matrice de ses valeurs.

Ex. 15 Parcours en largeur

Écrire une fonction `chemin g a b` qui reçoit une matrice représentant un graphe et qui renvoie un chemin sous forme de liste entre a et b . Ce chemin devra être celui que fournit le parcours en largeur.

Dans l'algorithme on modifie à chaque étape la fonction φ et donc la matrice du graphe associé à R_φ . On le fait en ajoutant ω à $\varphi(i, j)$ et en le retranchant à $\varphi(j, i)$ pour toute arête du chemin améliorant choisi ; ω est le minimum des poids des arêtes de ce chemin. On devra alors retrancher ω du poids de l'arête (i, j) et l'ajouter au poids de l'arête (j, i) dans la matrice du graphe.

Ex. 16 Modification des matrices

Écrire une fonction `modif_chemin g phi` qui reçoit une liste représentant un chemin dans un graphe g et une matrice représentant un flot et qui modifie les matrices g et phi comme indiqué ci-dessus.

Ex. 17 Recherche du flot

Écrire une fonction `flotMax g s p` qui reçoit une matrice représentant un graphe et le sommet et le puits et qui renvoie une fonction de flot maximal représentée par sa matrice.

Chapitre VII

Langages rationnels

1	Recherche d'un mot	110
1-1	Algorithme simple	110
1-2	Automates simples	111
2	Langages rationnels	112
2-1	Définitions	112
2-2	Produit	113
2-3	Étoile	114
2-4	Ensemble des langages rationnels	115
3	Expressions régulières	117
3-1	Arbre de construction	117
3-2	Langages réguliers	118
4	Langages locaux	121
5	Exercices	123
5-1	Langages	123
5-2	Expressions régulières	125
5-3	Langages locaux	126
5-4	Fonctions Caml	127

Un des objectifs de ce chapitre est d'introduire un formalisme efficace nous permettant de décrire certains ensembles de mots (autrement dit des langages) que l'on peut être amené à rechercher dans un texte. La caractéristique de ces langages est la possibilité de les décrire par des motifs (patterns en anglais), c'est-à-dire par des formules qu'on appelle, dans ce contexte, expressions régulières.

1 Recherche d'un mot

Le problème qui sert de cadre à ce chapitre est celui de la recherche d'un ou de plusieurs mots dans un texte. Dans le cas le plus simple on se donne un texte sous la forme d'une chaîne de caractères et une chaîne de caractère plus courte, le motif, dont on cherche les apparitions (les occurrences) dans le texte.

1-1 Algorithme simple

Nous avons déjà vu l'algorithme simple d'une telle recherche :

- ↪ si n est la longueur du texte et p celle de la chaîne recherchée
- ↪ on cherche, pour chaque position i possible (de 0 à $n - p$),
- ↪ si le mot cherché est égal à la portion de texte entre i et $i + p - 1$.

```
let rechercheTexte motif texte =  
  let n = string_length texte in  
  let p = string_length motif in  
  let resultat = ref [] in  
  for i = 0 to (n - p) do  
    if (sub_string texte i p) = motif  
      then result := i::(!resultat)  
  done;  
  !resultat;;
```

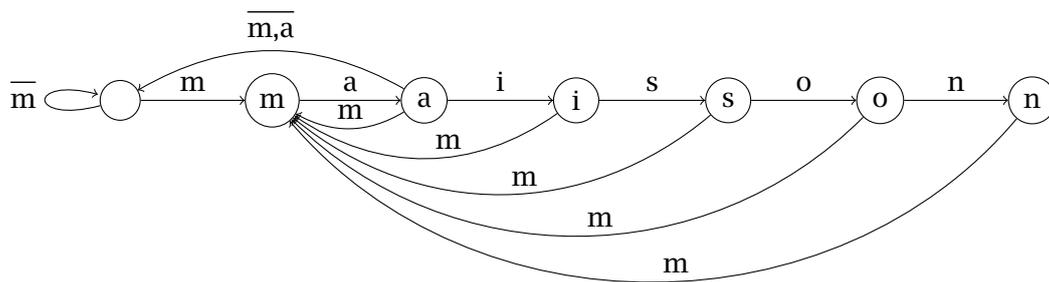
La fonction `sub_string` reçoit une chaîne de caractères `ch` et deux entiers `i` et `l` et renvoie la sous-chaîne de longueur `l` extraite de `ch` et commençant à l'indice `i`.

```
sub_string  
string -> int -> int -> string
```

1-2 Automates simples

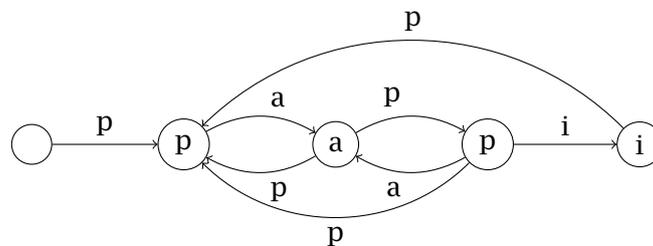
La méthode ci-dessus ci consiste à faire glisser une fenêtre sur le texte et à comparer les lettres vues dans la fenêtre avec celles du motif.

Il existe une autre méthode qui lit le texte lettre par lettre et dans laquelle on garde en mémoire le nombre de lettres égales au motif auquel on est parvenu. Dans l'exemple ci-dessous les flèches indiquent les changement d'état et sont marquées par la lettre qui implique le changement. La notation $\overline{\alpha}$ indique toute lettre autre que α de même $\overline{\alpha, \beta}$ indique les lettres qui ne sont ni α ni β .



On n'a pas indiqué les changement de i, s, o et n vers l'état initial marqués respectivement par $\overline{m, i}$, $\overline{m, s}$, $\overline{m, o}$ et $\overline{m, n}$.

L'automate peut être plus compliqué ; dans l'exemple suivant on n'indique pas les flèches vers l'état initial.



La situation se complique encore dans le cas de plusieurs mots.

Nous allons, dans la suite de ce chapitre, définir des ensembles de mots dont le chapitre suivant montrera qu'ils peuvent être reconnus par des automates.

2 Langages rationnels

Nous allons ici donner une définition très abstraites des ensembles de mots qu'il semble utile de rechercher. L'objectif sera de les construire de manière simple.

2-1 Définitions

1. Un **alphabet** est un ensemble fini.
Les éléments d'un alphabet sont les **lettres**.
Dans les exemples, pour simplifier, nous considérerons souvent un alphabet à 2 lettres $\mathcal{A} = \{a, b\}$.
2. Un **mot** est une suite finie de lettres (on dit aussi chaîne de caractères).
On notera un mot w par ses lettres : $w = u_1 u_2 \dots u_n$.
3. Le mot formé d'une lettre u répétée n fois est noté u^n .
4. L'ensemble des mots sur un alphabet \mathcal{A} est noté \mathcal{A}^+ .
5. Le nombre de lettres d'un mot w est sa longueur notée $|w|$.
L'ensemble des mots de longueur p se note \mathcal{A}^p .
6. Le nombre d'occurrences d'une lettre a dans un mot w est noté $|w|_a$.
On a $|w| = \sum_{a \in \mathcal{A}} |w|_a$.
7. On introduit le **mot vide**, il ne contient aucune lettre, sa longueur est nulle.
Il sera noté ε ou $1_{\mathcal{A}}$ (ou parfois Λ).
On définira $u^0 = \varepsilon$ pour toute lettre u .
8. L'ensemble des mots augmente de ε est noté $\mathcal{A}^* = \mathcal{A}^+ \cup \{\varepsilon\}$.
9. Un **langage** est un ensemble de mots, c'est une partie de \mathcal{A}^* .
10. Le complémentaire d'un langage L est la différence de \mathcal{A}^* et de L : $\bar{L} = \mathcal{A}^* \setminus L$.
11. Les **langages élémentaires** sont \emptyset , $\{\varepsilon\}$ et $\{u\}$ pour toute lettre u .

2-2 Produit

Produit

Le produit de deux mots est le mot obtenu par la concaténation de leurs lettres : si $u = u_1 \dots u_n$ et $v = v_1 \dots v_m$ alors $w = u.v = w_1 \dots w_{m+n}$ avec $w_i = u_i$ pour $1 \leq i \leq n$ et $w_i = v_{i-n}$ pour $n + 1 \leq i \leq n + m$.

On prolonge la définition par $w.\varepsilon = \varepsilon.w = w$.

On remarque qu'on a, pour tous $u, v \in \mathcal{A}^*$, $|u.v| = |u| + |v|$.

Monoïde

$(\mathcal{A}^*, .)$ est un monoïde, c'est-à-dire

- ↪ la loi est associative
- ↪ elle admet un élément neutre, ici ε .

Comme toutes les possibilités de calculs donnent le même résultat on pourra omettre les parenthèses et écrire $u.v.w$ pour $(u.v).w$ ou $u.(v.w)$. Un monoïde est la structure la plus simple : $(\mathbb{N}, +)$ est un monoïde, ainsi que $(A, *)$ si $(A, +, *)$ est un anneau.

Simplifications

- ↪ Si $u.v = u.v'$ alors $v = v'$.
- ↪ Si $u.v = u'.v$ alors $u = u'$.

Produit de langages

Le produit des langages L_1 et L_2 est le langage $L_1.L_2$ défini par
 $L_1.L_2 = \{u_1.u_2 ; u_1 \in L_1, u_2 \in L_2\}$

Exemples

- ↪ $\emptyset.L = L.\emptyset = \emptyset$.
- ↪ $\{\varepsilon\}.L = L.\{\varepsilon\} = L$.
- ↪ $\{u\}.\{v\} = \{u.v\}$.
- ↪ $\mathcal{A}^+ = \mathcal{A}.\mathcal{A}^*$.
- ↪ L'ensemble des mots de \mathcal{A} qui commencent par a est $\{a\}.\mathcal{A}^*$.
- ↪ $\mathcal{A}^*.\{a\}.\mathcal{A}^*$ est le langage des mots contenant au moins une fois la lettre a
- ↪ $\{a\}.\{b, \varepsilon\}.\{a, b\} = \{a^2, ab, aba, ab^2\}$.
- ↪ Ne pas confondre $L.L$ et $\{w.w ; w \in L\}$.

2-3 Étoile

Si L est un langage on définit $L^0 = \{\varepsilon\}$ et, par récurrence, $L^{n+1} = L^n.L$ pour $n \geq 1$.

- ↪ $L^1 = L$
- ↪ $L^n = \{w_1.w_2 \dots w_n ; w_i \in L \text{ pour tout } i\}$.
- ↪ $L^n.L^m = L^{n+m}$

Étoile

L'étoile d'un langage L est $L^* = \bigcup_{n \in \mathbb{N}} L^n$.
 On définit aussi $L^+ = \bigcup_{n \in \mathbb{N}^*} L^n$

- ↪ L^* est l'ensemble des produits de mots de L .
- ↪ Si $L = \mathcal{A}$ on retrouve bien l'ensemble de tous les mots : la notation \mathcal{A}^* est non-ambiguë.
- ↪ Si $L = \{w\}$ alors $L^* = \{w^n ; n \in \mathbb{N}\}$.
- ↪ $\emptyset^* = \{\varepsilon\}^* = \{\varepsilon\}$.
- ↪ Si L contient un mot non vide alors L^* est infini.

2-4 Ensemble des langages rationnels

Un produit d'un nombre fini de langages élémentaires est un langage vide ou réduit à un mot.

Une union finie de tels langages permet de construire tout langage fini.

On va aller au-delà des langages finis en s'intéressant aux langages que l'on peut construire à partir des langages élémentaires à l'aide d'unions, de produits et d'étoiles de langages.

Ensembles des langages rationnels

On définit par récurrence

$$\mathcal{R}_0 = \{\emptyset, \{\varepsilon\}\} \cup \bigcup_{a \in \mathcal{A}} \{a\} \text{ et}$$

$$\mathcal{R}_{n+1} = \{L_1 \cup L_2 \mid L_1, L_2 \in \mathcal{R}_n\} \cup \{L_1.L_2 \mid L_1, L_2 \in \mathcal{R}_n\} \cup \{L^* \mid L \in \mathcal{R}_n\}$$

$$\text{L'ensemble des langages rationnels est } \mathcal{R} = \bigcup_{n \in \mathbb{N}} \mathcal{R}_n.$$

\mathcal{R} contient tous les langages élémentaires

\mathcal{R} est stable par union produit et étoile.

Démonstration

Pour tout $L \in \mathcal{R}_n$ on a $L = L \cup L \in \mathcal{R}_{n+1}$ donc $\mathcal{R}_n \subset \mathcal{R}_{n+1}$. Ainsi \mathcal{R} contient \mathcal{R}_0 donc tous les langages élémentaires.

Si $L_1, L_2 \in \mathcal{R}$ alors il existe des entiers n et m tels que $L_1 \in \mathcal{R}_n$ et $L_2 \in \mathcal{R}_m$ donc $L_1, L_2 \in \mathcal{R}_p$ avec $p = \max(n, m)$.

On a alors $L_1 \cup L_2 \in \mathcal{R}_{p+1} \subset \mathcal{R}$, $L_1.L_2 \in \mathcal{R}_{p+1} \subset \mathcal{R}$ et $L_1^* \in \mathcal{R}_{p+1} \subset \mathcal{R}$.

D'après les remarques ci-dessus, **tout langage fini est rationnel.**

Caractérisation

Si un ensemble \mathcal{L} de langages (c'est-à-dire une partie de $\mathcal{P}(\mathcal{A}^*)$)

- contient tous les langages élémentaires
- est stable par union, produit et étoile

alors $\mathcal{R} \subset \mathcal{L}$.

Démonstration

La démonstration se fait en montrant, par récurrence sur n , que \mathcal{R}_n est inclus dans \mathcal{L} pour tout n .

1. La première propriété implique $\mathcal{R}_0 \subset \mathcal{L}$.

2. Si on suppose qu'on a $\mathcal{R}_n \subset \mathcal{L}$ alors la stabilité de \mathcal{L} par les opérations implique qu'on a $\{L_1 \cup L_2 \mid L_1, L_2 \in \mathcal{R}_n\} \subset \mathcal{L}$, $\{L_1.L_2 \mid L_1, L_2 \in \mathcal{R}_n\} \subset \mathcal{L}$ et $\{L^* \mid L \in \mathcal{R}_n\} \subset \mathcal{L}$ d'où $\mathcal{R}_{n+1} \subset \mathcal{L}$

On en déduit que $\mathcal{R} = \bigcup_{n \in \mathbb{N}} \mathcal{R}_n \subset \mathcal{L}$.

On peut traduire cette propriété en disant que \mathcal{R} est le plus petit ensemble de langages contenant tous les langages élémentaires et stable par union, produit et étoile.

Une conséquence utile de cette caractérisation est la propriété suivante.

Induction structurelle

Si une propriété P portant sur des langages est telle que

- $P(\emptyset)$, $P(\{\varepsilon\})$ et $P(\{a\})$ pour $a \in \mathcal{A}$ sont vérifiées
 - si $P(L_1)$ et $P(L_2)$ sont vraies
alors $P(L_1 \cup L_2)$, $P(L_1.L_2)$ et $P(L_1^*)$ sont vraies
- alors $P(L)$ est vraie pour tout langage rationnel.

Il suffit de considérer l'ensemble \mathcal{L} des langages L tels que $P(L)$ est vérifiée.

3 Expressions régulières

Dans cette partie nous allons donner, pour chaque langage rationnel, une construction à partir des langages élémentaires. L'alphabet \mathcal{A} est fixé.

3-1 Arbre de construction

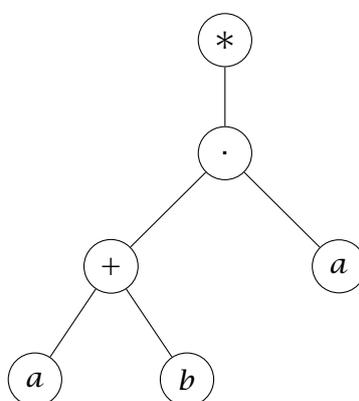
On considère les arbres dont les feuilles sont des langages élémentaires, on représentera le vide par $\mathbf{0}$ et ϵ par $\mathbf{1}$, et les nœuds internes sont

- ↪ soit un nœud binaire signifiant l'union, notée ici $+$,
- ↪ soit un nœud binaire signifiant le produit,
- ↪ soit un nœud unaire signifiant l'étoile.

Un arbre permet donc de construire un langage à partir de langages élémentaires à l'aide d'union, de produits et d'étoiles ; on obtient ainsi un langage rationnel.

Inversement une démonstration par récurrence sur n montre que tout élément de \mathcal{R}_n est constructible à partir d'un arbre de hauteur n au plus donc tout langage rationnel est constructible par un arbre.

Par exemple le langage $\{ba, aa\}^*$ peut être construit par $((\{a\} \cup \{b\}) \cdot \{a\})^*$ donc on peut lui associer l'arbre



On considère qu'un tel arbre est un arbre binaire pour lequel le fils unique d'une étoile est considéré comme un fils gauche.

Le parcours infixe parenthésé de cet arbre est une chaîne de caractère : l'arbre ci-dessus donne $((a+b) \cdot a)^*$.

On va donner une expression formelle de cette chaîne de caractère.

3-2 Langages réguliers

Expression régulière

\mathcal{A} est un alphabet ne contenant pas les lettres "0", "1", "*", "+", ".", "(", ")". Une expression régulière sur \mathcal{A} est une expression de la forme

- ↪ 0,
- ↪ 1,
- ↪ a avec $a \in \mathcal{A}$,
- ↪ (r_1+r_2) avec r_1 et r_2 expressions régulières,
- ↪ $(r_1.r_2)$ avec r_1 et r_2 expressions régulières ou
- ↪ (r^*) avec r expression régulière.

Une expression régulière fournit une construction d'un langage.

Langage associé

À toute expression régulière e sur \mathcal{A} on peut associer un langage par la construction récursive

- ↪ $L[0] = \emptyset$,
- ↪ $L[1] = \{\varepsilon\}$,
- ↪ $L[a] = \{a\}$ pour $a \in \mathcal{A}$,
- ↪ $L[(r_1 + r_2)] = L[r_1] \cup L[r_2]$,
- ↪ $L[(r_1.r_2)] = L[r_1].L[r_2]$,
- ↪ $L[(r^*)] = (L[r])^*$.

Un tel langage est appelé régulier.

Un langage régulier est construit à partir de langages élémentaires par des unions, produits et étoiles ; il est donc rationnel. Inversement le passage par l'arbre de construction montre qu'un langage rationnel peut être défini par une expression régulière.

Dénotation

Les langages rationnels sont les langages réguliers.
Si $L = L[r]$ on dit que L est **dénoté** par l'expression régulière r .

Il faut noter que si le langage dénoté par une expression régulière est unique il peut exister plusieurs expressions régulières qui dénotent un langage donné.

Par exemple le langage donné en exemple est dénoté, comme on l'a vu, par $((a+b) \cdot a)^*$ mais aussi par $((b \cdot a + a \cdot a)^*)$ ou $((a \cdot a)^* \cdot ((b \cdot a) \cdot ((a \cdot a)^*))^*)$.

On voit que les notations deviennent vite lourdes : on pourra éliminer des parenthèses en utilisant la priorité de l'étoile sur le produit et la priorité du produit sur la somme. Les expressions régulières ci-dessus deviennent $((a+b) \cdot a)^*$, $(b \cdot a + a \cdot a)^*$ et $(a \cdot a)^* \cdot (b \cdot a \cdot (a \cdot a)^*)^*$.

Expressions équivalentes

Deux expressions régulières sont équivalentes si elles dénotent le même langage.

Parmi les langages élémentaires nous avons ajouté \emptyset et $\{\varepsilon\}$. Nous allons voir qu'en fait ils servent uniquement à fabriquer l'ensemble vide ou à ajouter le mot vide.

Théorème : élimination de zéro

Tout langage rationnel non vide est dénoté par une expression régulière ne contenant pas \emptyset .

Démonstration : elle se fait par induction structurelle.

$P(L)$ est la propriété :

L est vide ou est dénoté par un expression régulière ne contenant pas \emptyset .

1. Pour les langages élémentaires la propriété est évidente.
2. On suppose que $P(L_1)$ et $P(L_2)$ sont vraies.
 \rightarrow Si L_1 est vide alors $L_1 \cup L_2 = L_2$ donc $P(L_1 \cup L_2)$ est vraie

↪ De même si L_2 est vide
 ↪ Si L_1 et L_2 sont non vides alors ils sont dénotés par des expressions régulières r_1 et r_2 ne contenant pas \emptyset . $L_1 \cup L_2$ est alors dénoté par (r_1+r_2) qui ne contient pas \emptyset .
 Ainsi $P(L_1 \cup L_2)$ est vraie.

3. On suppose que $P(L_1)$ et $P(L_2)$ sont vraies.

↪ Si L_1 ou L_2 est vide alors $L_1.L_2$ est vide donc $P(L_1.L_2)$ est vraie
 ↪ Si L_1 et L_2 sont non vides alors ils sont dénotés par des expressions régulières r_1 et r_2 ne contenant pas \emptyset . L_1L_2 est alors dénoté par $(r_1.r_2)$ qui ne contient pas \emptyset .
 Ainsi $P(L_1 \cup L_2)$ est vraie.

4. On suppose que $P(L)$ est vraie.

↪ Si L est vide alors $L^* = \{\varepsilon\}$ est dénoté par $\mathbf{1}$ donc $P(L^*)$ est vraie.
 ↪ Si L est non vide alors il est dénoté par une expression régulière r ne contenant pas \emptyset . L^* est alors dénoté par (r^*) qui ne contient pas \emptyset . Ainsi $P(L^*)$ est vraie.

Une formulation alternative est que toute expression régulière est équivalente à \emptyset ou à une expression régulière ne contenant pas \emptyset .

Une expression régulière est appelée réduite si elle ne contient ni \emptyset ni $\mathbf{1}$.

Théorème : élimination du un

Tout langage rationnel est dénoté par une expression régulière de la forme \emptyset , $\mathbf{1}$, r ou $(r+\mathbf{1})$ avec r expression rationnelle réduite.

La démonstration est proposée dans l'exercice **11**

4 Langages locaux

Nous allons, dans cette partie, définir un type de langages caractérisé par une lecture locale des lettres (d'où le nom). Le principe est de ne lire les mots (et tester les lettres) qu'avec une fenêtre de lecture de largeur 2.

Langage local

Un langage L sur l'alphabet \mathcal{A} est local s'il existe $P \subset \mathcal{A}$, $S \subset \mathcal{A}$ et $F \subset \mathcal{A}^2$ tels que $u = u_1u_2\cdots u_n \in L \setminus \{\varepsilon\}$ est équivalent à $u_1 \in P$, $u_n \in S$ et $u_iu_{i+1} \in F$ pour tout $i \in \{1, 2, \dots, n-1\}$.

L'appartenance de ε à L est facultative ; en plus des 3 ensembles il faut donc un quatrième déterminant pour indiquer si on a $\varepsilon \in L$.

Exemples

- ↪ \emptyset et $\{\varepsilon\}$ sont locaux : ils sont définis par $P = F = S = \emptyset$.
- ↪ $L = \{a\}$ avec $a \in \mathcal{A}$ est local : $P = S = \{a\}$, $F = \emptyset$, $\varepsilon \notin L$.
- ↪ $L = \{(ab)^n ; n \in \mathbb{N}\}$ est local, défini par $P = \{a\}$, $F = \{ab, ba\}$, $S = \{b\}$ et $\varepsilon \in L$.
Si on retire ε on obtient $\{(ab)^n ; n \in \mathbb{N}^*\}$.
- ↪ $L = \{a^n b ; n \in \mathbb{N}^*\}$ est local, défini par $P = \{a\}$, $F = \{ab, aa\}$, $S = \{b\}$ et $\varepsilon \notin L$.
- ↪ $L = \{a^n b^p ; n, p \in \mathbb{N}\}$ est local, défini par $P = \{a, b\}$, $F = \{aa, ab, bb\}$, $S = \{a, b\}$ et $\varepsilon \in L$.

On peut noter que

1. l'ensemble des préfixes de taille 1 de L est inclus dans P ,
2. l'ensemble des facteurs de taille 2 de L est inclus dans F et
3. l'ensemble des suffixes de taille 1 de L est inclus dans S .

Cependant ces inclusions ne sont pas forcément des égalités. Par exemple, pour $\mathcal{A} = \{a, b\}$, $L = \{a^n ; n \in \mathbb{N}\}$ est local, défini par $P = \{a\}$, $F = \{aa\}$, $S = \{a\}$ et $\varepsilon \in L$ mais il peut être aussi défini par $P = \{a, b\}$, $F = \{aa, ab\}$, $S = \{a\}$ et $\varepsilon \in L$.

On peut remplacer l'ensemble F par l'ensemble $N = \mathcal{A}^2 \setminus F$, la condition sur les facteurs devient $u_iu_{i+1} \notin N$.

Cela permet de donner une définition plus synthétique des langages locaux.

Caractérisation des langages locaux

Un langage L sur l'alphabet \mathcal{A} est local s'il existe $P \subset \mathcal{A}$, $S \subset \mathcal{A}$ et $N \subset \mathcal{A}^2$ tels que $L \setminus \{\varepsilon\} = (P.\mathcal{A}^* \cap \mathcal{A}^*.S) \setminus \mathcal{A}^*.N.\mathcal{A}^*$.

La caractérisation permet de construire un langage local à partir des langages rationnels ($P.\mathcal{A}^*$, $\mathcal{A}^*.S$ et $\mathcal{A}^*.N.\mathcal{A}^*$) mais on utilise des opérations non rationnelles : l'intersection et la différence. On ne peut pas conclure sur la rationalité de ces langages avant les résultats du chapitre suivant.

On verra en exercice qu'il existe des langages rationnels qui ne sont pas locaux. Cependant il existe un cas particulier qui sera utilisé dans le chapitre suivant.

Théorème : expressions linéaires

Une expression régulière est linéaire si toutes les lettres qui la composent sont distinctes.
Si e est une expression régulière linéaire alors L_e est un langage local.

La démonstration se fait par induction structurale sur les expressions régulières en utilisant le lemme suivant. Voir l'exercice 22

Lemme : union et produits sur des langages disjoints

L'alphabet \mathcal{A} est partitionné en $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ avec $\mathcal{A}_1 \cap \mathcal{A}_2 = \emptyset$.
Si L_1 est un langage local sur \mathcal{A}_1 et L_2 est un langage local sur \mathcal{A}_2
alors $L_1 \cup L_2$ et $L_1.L_2$ sont locaux.

La démonstration est proposée dans l'exercice 21

5 Exercices

- Ex. 1 Complexité** Déterminer, en nombre de comparaisons de deux caractères, la complexité de rechercheTexte ; on négligera la complexité de sub_string. Donner un exemple où la complexité atteint son maximum.

5-1 Langages

- Ex. 2 Alphabet à une lettre**

On considère $\mathcal{A} = \{a\}$ et $L_k = \{a^{3n+k}, n \in \mathbb{N}\}$ pour $k \in \mathbb{N}$.

Déterminer $L_k \cap L_j$.

Déterminer $L_k.L_j$.

- Ex. 3 Associativité**

Prouver l'associativité du produit de mots.

En déduire que pour des langages L_1, L_2 et L_3 on a $(L_1.L_2).L_3 = L_1.(L_2.L_3)$.

- Ex. 4 Préfixes** Un mot v est un **préfixe** (resp. suffixe) de u s'il existe w tel que $u = v.w$ (resp. $u = w.v$) ; on note $v < u$. Prouver

$\rightarrow v < u$ si et seulement si $|v| \leq |u|$ et $v_i = u_i$ pour $1 \leq i \leq |v|$.

$\rightarrow <$ est une relation d'ordre.

- Ex. 5 Puissances** Pour tout mot w on note $w^0 = \varepsilon$ et, par récurrence, $w^{n+1} = w^n.w$.

En particulier $w^1 = w$. Prouver que $w^n.w^p = w^p.w^n = w^{n+p}$.

- Ex. 6 Exemples de langages rationnels**

1. Prouver que l'ensemble des mots de longueur n au plus est rationnel.
2. Prouver que l'ensemble des mots de longueur n au moins est rationnel.
3. Prouver que l'ensemble des mots qui commencent et qui finissent par la même lettre est rationnel.

Ex. 7 Racine d'un langage

Pour un langage L , sa racine est $\sqrt{L} = \{u \in \Sigma / u.u \in L\}$.

Déterminer $\sqrt{L_k}$ où $L_k = \{a^{3n+k}, n \in \mathbb{N}\}$.

Montrer que $\epsilon \in L$ si et seulement si $\epsilon \in \sqrt{L}$.

Ex. 8 Code

Un code sur un alphabet \mathcal{A} est un langage L sur \mathcal{A} tel que l'égalité $x_1.x_2.\dots.x_p = y_1.y_2.\dots.y_q$ avec $x_1, \dots, x_p, y_1, \dots, y_q$ dans L entraîne $p = q$ et $x_i = y_i$ pour tout i .

1. Déterminer les codes parmi les langages finis suivants :

$$\hookrightarrow L_1 = \{ab, baa, abba, aabaa\},$$

$$\hookrightarrow L_2 = \{b, ab, baa, abaa, aaaa\},$$

$$\hookrightarrow L_3 = \{aa, ab, aab, bba\},$$

$$\hookrightarrow L_4 = \{a, ba, bba, baab\}.$$

2. Soit $u \in \mathcal{A}^*$, montrer que $\{u\}$ est un code si et seulement si $u \neq \epsilon$.

3. Soit u et v deux mots distincts ; montrer que $\{u, v\}$ est un code si et seulement si $u.v \neq v.u$.

4. Soit L un langage ne contenant pas ϵ et tel qu'aucun mot de L ne soit préfixe d'un autre mot de L . Montrer que L est un code.

Ex. 9 Lemme d'Arden A et B sont deux langages sur un même alphabet. On suppose que L vérifie $L = A.L \cup B$.

1. Montrer que $A^*.B \subset L$.

2. Montrer que si $\epsilon \notin A$ alors $L = A^*.B$.

3. Montrer que si $\epsilon \in A$ alors les solutions sont les ensembles de la forme $A^*.C$ avec C contenant B .

4. **Application.** Sur $\mathcal{A} = \{a, b\}$, on note L_1 le langage des mots ayant un nombre pair de b et L_2 le langage des mots ayant un nombre impair de b . Écrire deux relations linéaires liant L_1 et L_2 puis utiliser le lemme d'Arden pour en donner une expression.

Ex. 10 Transformations L est un langage rationnel sur un alphabet \mathcal{A} .

On suppose que a est une lettre de \mathcal{A} .

Prouver que chacun des langages suivants est rationnel.

1. Le langage des mots préfixes des mots de L .
2. L'ensemble des mots de L qui ne contiennent pas a .
3. L'ensemble des mots de L qui contiennent a .
4. L'ensemble des mots obtenus en enlevant le premier a dans les mots de L contenant a .
5. L'ensemble des mots obtenus en enlevant un a dans les mots de L contenant a .

5-2 Expressions régulières

Ex. 11 Réduction

Prouver que tout langage rationnel est dénoté par une expression régulière de la forme $\mathbf{0}$, $\mathbf{1}$, r ou $(r+1)$ avec r expression rationnelle ne contenant ni $\mathbf{0}$ ni $\mathbf{1}$.

Ex. 12 Détermination d'expressions régulières

1. L'ensemble des mots qui contiennent au moins un a ,
2. L'ensemble des mots qui contiennent au plus un a ,
3. L'ensemble des mots tels que toute série de a soit de longueur paire,
4. L'ensemble des mots dont la longueur n'est pas divisible par 3,
5. L'ensemble des mots tels que deux lettres consécutives soient toujours distinctes,
6. L'ensemble des mots contenant au moins un a et un b .

Ex. 13 3 lettres Donner une expression régulière dénotant l'ensemble L des mots sur l'alphabet $\mathcal{A} = \{a, b, c\}$ tels que deux lettres consécutives soient toujours distinctes.

Ex. 14 Complémentaire Donner une représentation régulière dénotant \bar{L} avec L dénoté par $(a+b)^* . a . b . a . (a+b)^*$.

Ex. 15 Expressions équivalentes

1. Montrer que $(a+b)^*$, $(a^*.b)^*.a^*$, $a^*+a^*.b.(a+b)^*$ et $b^*. (a.a^*.b.b^*)^*.a^*$ sont équivalentes.
2. Prouver que $(r1.r2)^*$ et $1+r1.(r2.r1)^*.r2$ sont équivalentes.

5-3 Langages locaux**Ex. 16 Définitions multiples** Donner 3 autres définitions locales de $L = \{a^n ; n \in \mathbb{N}\}$ **Ex. 17 Caractérisation**

L est un langage local sur \mathcal{A} , défini par P, F et S ; on note $N = \mathcal{A}^2 \setminus F$.
Montrer que $L \setminus \{\varepsilon\} = (P.\mathcal{A}^* \cap \mathcal{A}^*.S) \setminus \mathcal{A}^*.N.\mathcal{A}^*$.

Ex. 18 Intersection de langages locaux

Prouver que l'intersection de deux langages locaux sur \mathcal{A} est locale.

Ex. 19 Étoile d'un langage local

Prouver que l'étoile d'un langage local sur \mathcal{A} est locale.

Ex. 20 Contre-exemple On pose $L_1 = \{ab\}$ et $L_2 = \{a^n ; n \in \mathbb{N}\}$.

Prouver que L_1 et L_2 sont locaux mais que ni $L_1 \cup L_2$ ni $L_1.L_2$ ne sont locaux.
On a donc des exemples de langages rationnels non locaux.

Ex. 21 Alphabets distincts Montrer que si L_1 est un langage local sur \mathcal{A}_1 et L_2 est un langage local sur \mathcal{A}_2 avec $\mathcal{A}_1 \cap \mathcal{A}_2 = \emptyset$ alors $L_1 \cup L_2$ et $L_1.L_2$ sont locaux.**Ex. 22 Localité des langages définis par un expression régulière linéaire**

Montrer que si r est une expression régulière linéaire alors $L[r]$ est un langage local.

5-4 Fonctions Caml

Le type des expressions régulières est semblable à celui des arbres

```
type expreg = |Zero
              |Un
              |Lettre of string
              |Somme of expreg * expreg
              |Produit of expreg * expreg
              |Etoile of expreg;;
```

Ex. 23 Langage vide

Écrire une fonction `lVide` qui renvoie un booléen qui répond à la question :
"Le langage est-il vide ?"
pour un langage dénoté par une expression régulière.

Ex. 24 Mot vide

Écrire une fonction `motVide` qui renvoie un booléen qui répond à la question :
"Le langage contient-il le mot vide ?"
pour un langage dénoté par une expression régulière.

Ex. 25 Vers les langages locaux Écrire des fonctions `prefixes`, `suffixes` et `facteurs` qui calculent les ensembles des préfixes de taille 1, des suffixes de taille 1 et des facteurs de taille 2 d'un langage dénoté par une expression régulière.

Chapitre **VIII**

Automates

1	Automates déterministes	130
1-1	Transitions	130
1-2	Automates déterministes complets	132
1-3	Émondage	134
1-4	Opérations ensemblistes	135
1-5	Automates déterministes incomplets	138
2	Automates non-déterministes	141
2-1	Définitions	141
2-2	Déterminisation	143
3	L'algorithme de Berry-Sethi	145
3-1	Automate local	145
3-2	Morphismes	146
3-3	Construction de l'automate	147
3-4	Un exemple	148
4	Exercices	151
4-1	Automates déterministes	151
4-2	Automates non déterministes	152
4-3	Reconnaissabilité des langages rationnels	153

Nous avons déjà rencontré un automate dans un chapitre précédent sans le définir. De manière informelle, un automate est une machine abstraite qui peut prendre un nombre fini d'états, et qui reçoit en entrée des commandes qui déterminent un changement d'état. Ce comportement est idéal, nous allons donner des généralisations où les commandes peuvent ne pas être complètes voire ne pas induire un comportement unique. En isolant des états de départ et d'arrivée nous associerons un langage à chaque automate. Un résultat important sera que les généralisations des définitions ne définissent pas des langages plus généraux.

1 Automates déterministes

1-1 Transitions

La première étape va être de donner un cadre aux commandes et états.

Machine

Une machine est un triplet (\mathcal{A}, S, δ) où

1. \mathcal{A} est un ensemble fini, l'alphabet, ses éléments seront les lettres,
2. S est un ensemble fini, l'ensemble des états,
3. δ est une fonction de $S \times \mathcal{A}$ vers S , la fonction de transition.

Une machine est caractérisée par la fonction de transition ; en effet l'alphabet et l'ensemble des états sont les composantes de son ensemble de départ. Voici un exemple.

	0	1	2
a	1	1	2
b	0	2	2

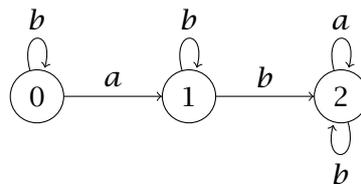
Transition

Une transition d'une machine (\mathcal{A}, S, δ) est un triplet (s, a, s') tel que $\delta(s, a) = s'$.
 On notera aussi $s.a = s'$ ou $s \xrightarrow{a} s'$.
 s est l'origine de la transition, s' en est l'extrémité et a son étiquette.

Une représentation peut être faite par un graphe valué par \mathcal{A} : S est l'ensemble des sommets et les arêtes sont les transitions.

Contrairement aux graphes vus jusqu'à présent les arêtes d'un sommet vers lui-même sont possibles et il peut y avoir plusieurs arêtes d'un sommet s vers un sommet s' : elles auront alors des étiquettes distinctes.

On peut noter que, si n est le cardinal de \mathcal{A} , il y a n arêtes sortantes depuis tout sommet, étiquetées par les n lettres de l'alphabet.



Une suite finie de commandes est une suite de lettres donc un mot.

On peut associer une transition à un mot en composant les transition des lettres : on fait le choix de lire les lettres de la gauche vers la droite.

Action d'un mot

Si $M = (\mathcal{A}, S, \delta)$ est une machine sur le langage \mathcal{A} on prolonge la fonction de transition à \mathcal{A}^* tout entier par la fonction δ^* de $S \times \mathcal{A}^*$ vers S avec

1. $\delta^*(s, \varepsilon) = s$ pour tout $s \in S$
2. $\delta^*(s, u.x) = \delta(\delta^*(s, u), x)$ pour $u \in \mathcal{A}^*$ et $x \in \mathcal{A}$.

On notera encore $\delta^*(s, u) = s.u$ ou $s \xrightarrow{u} s'$.

On peut donc écrire $s.(u.x) = (s.u).x$ pour $u \in \mathcal{A}^*$ et $x \in \mathcal{A}$.

Pour $u = x_1x_2\cdots x_n$, on a $s.u = ((\cdots((s.x_1).x_2).\cdots.x_{n-1}).x_n)$.

Si on pose $s_1 = \delta(s, x_1)$, $s_2 = \delta(s_1, x_2)$, ..., $s' = s_n = \delta(s_{n-1}, x_n)$ alors $(s, s_1, s_2, \dots, s_n)$ est le **parcours** (ou **calcul**) de u à partir de s .

On notera ce parcours sous la forme

$$s \xrightarrow{x_1} s_1 \xrightarrow{x_2} s_2 \xrightarrow{x_3} \cdots \xrightarrow{x_{n-1}} s_{n-1} \xrightarrow{x_n} s_n$$

Produit

$s.(u.v) = (s.u).v$ pour tous $u, v \in \mathcal{A}^*$.

1-2 Automates déterministes complets

Pour l'instant les parcours de la machine se font sans condition de point de départ et tous les parcours ont la même valeur. Nous allons enrichir la machine en lui ajoutant un point de départ et des sommets d'arrivée souhaités.

Automate déterministe complet

Un automate fini déterministe ou A.F.D. (DFA en anglais) est un quintuplet $Q = (\mathcal{A}, S, \delta, s_0, T)$ où

1. \mathcal{A} est un alphabet (fini),
2. S est un ensemble fini d'états,
3. δ est une application de $S \times \mathcal{A}$ vers S : la fonction de transition.
4. s_0 est un élément de S , l'état initial,
5. T est une partie de S , l'ensemble des états finaux, (on dit aussi accepteurs ou terminaux).

Les 3 premiers élément du quintuplet forment donc une machine.

Un automate va servir à tester les mots : a-t-on $s_0.u \in T$?

On dit alors que u est **reconnu** par l'automate.

Langage reconnu

1. Le langage reconnu (ou accepté) par un automate $Q = (\mathcal{A}, S, \delta, s_0, T)$ est l'ensemble $L(Q)$ de tous les mots reconnus par Q .

$$L(Q) = \{u \in \mathcal{A}^* ; s_0.u \in T\}$$

2. Un langage L est **reconnaisable** s'il existe un automate déterministe complet Q tel que $L = L(Q)$.

$Q = (\mathcal{A}, S, \delta, s_0, T)$ est un automate.

↪ Si $T = \emptyset$ alors $L(Q) = \emptyset$.

↪ Si $T = S$ alors $L(Q) = \mathcal{A}^*$.

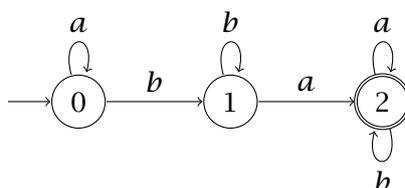
↪ $L(Q)$ contient ε si et seulement si s_0 appartient à T .

Pour représenter graphiquement un automate

↪ l'état initial est indiqué par une flèche entrante

↪ un état terminal est entouré d'un double cercle ou est marqué par une flèche sortante.

Dans l'exemple donné ci-dessus, on pose $s_0 = 0$ et $T = \{2\}$.

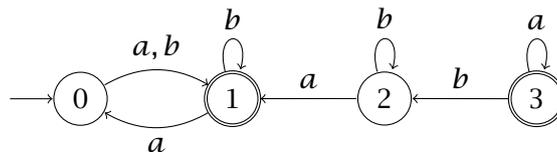


Le langage reconnu est l'ensemble des mots contenant ba .

1-3 Émondage

Nous allons dans la suite déterminer des automates différents en conservant le langage. La première possibilité est d'enlever les états non utilisés.

Dans l'automate ci-dessous on voit qu'aucun parcours ne passera par les états 2 ou 3.



On va essayer de les retirer.

Accessibilité

$Q = (\mathcal{A}, S, \delta, s_0, T)$ est un automate déterministe complet.
Un état $s \in S$ est dit accessible s'il existe $u \in \mathcal{A}^*$ tel que $s = s_0.u$.

Les états accessibles sont les seuls états visibles depuis s_0 .

États accessibles

Si S' est l'ensemble des états accessibles d'un automate $Q = (\mathcal{A}, S, \delta, s_0, T)$ alors, pour tout $s \in S'$ et pour tout $a \in \mathcal{A}$, $\delta(s, a)$ appartient à S' .

Démonstration : tout élément s de S' est de la forme $s = s_0.v$ donc

$$\delta(s, a) = s.a = (s_0.u).a = s_0.(u.a) \in S'.$$

Ainsi la restriction de δ à $S' \times \mathcal{A}$ est à valeur dans S' , on la note encore δ .

On obtient ainsi un nouvel automate

Automate émondé

Si S' est l'ensemble des états accessibles d'un automate $Q = (\mathcal{A}, S, \delta, s_0, T)$, l'automate émondé associé à Q est l'automate déterministe complet

$$Q' = (\mathcal{A}, S', \delta, s_0, T \cap S')$$

On a $L(Q) = L(Q')$.

Démonstration :

↪ Si u appartient à $L(Q')$ alors $s_0.u \in T \cap S' \subset T$ donc $u \in L(Q) : L(Q') \subset L(Q)$.

↪ Si u appartient à $L(Q)$ alors $s_0.u \in T$.

Les sommets atteints depuis s_0 appartiennent à S' donc le chemin est un chemin dans S' .

En particulier $s_0.u \in S'$ donc $s_0.u \in T \cap S'$ et $u \in L(Q')$.

Ainsi $L(Q') \subset L(Q)$ donc $L(Q') = L(Q)$.

1-4 Opérations ensemblistes

Nous allons prouver ici quelques stabilités des langages reconnaissables.

On rappelle que le complémentaire d'un langage L sur \mathcal{A} est $\bar{L} = \mathcal{A}^* \setminus L$.

Complémentaire

Le complémentaire d'un langage reconnaissable est reconnaissable.

Démonstration : soit L reconnaissable. On considère un automate $Q = (\mathcal{A}, S, \delta, s_0, T)$ qui le reconnaît.

On a alors $u \in \bar{L}$ équivalent à $s_0.u \notin T$, c'est-à-dire à $s_0.u \in \bar{T} = S \setminus T$.

Ainsi \bar{L} est reconnu par l'automate $(\mathcal{A}, S, s_0, \delta, \bar{T})$.

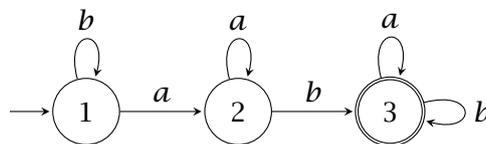
Pour l'union et le produit nous aurons besoin d'une construction de machine.

Produit de machines

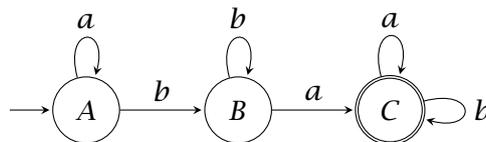
Si $M_1 = (\mathcal{A}, S_1, \delta_1)$ et $M_2 = (\mathcal{A}, S_2, \delta_2)$ sont deux machines sur le même alphabet leur produit $M_1 \odot M_2$ est la machine $(\mathcal{A}, S_1 \times S_2, \delta_1 \times \delta_2)$ avec $\delta_1 \times \delta_2((s, t), a) = (\delta_1(s, a), \delta_2(t, a))$.

Ainsi, en notant de la même façon l'action des lettres dans M_1, M_2 et $M_1 \odot M_2$, on a $(s, t).x = (s.x, t.x)$ pour toute lettre de \mathcal{A} d'où, par récurrence, pour tout mot de \mathcal{A}^* , $(s, t).u = (s.u, t.u)$.

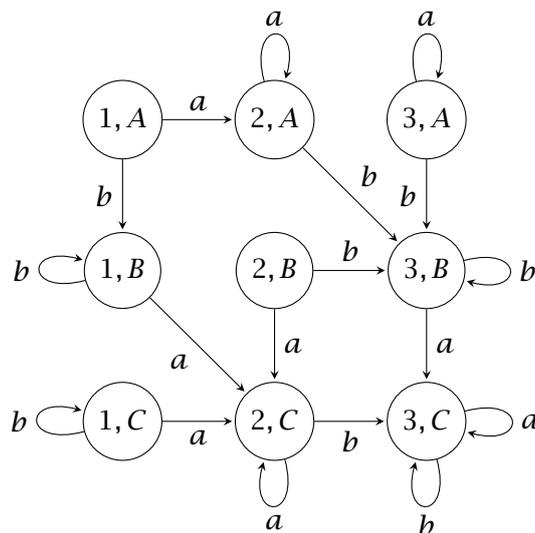
Exemple L'automate suivant reconnaît L_1 , ensemble des mots contenant ab .



L'automate suivant reconnaît L_2 , ensemble des mots contenant ba .

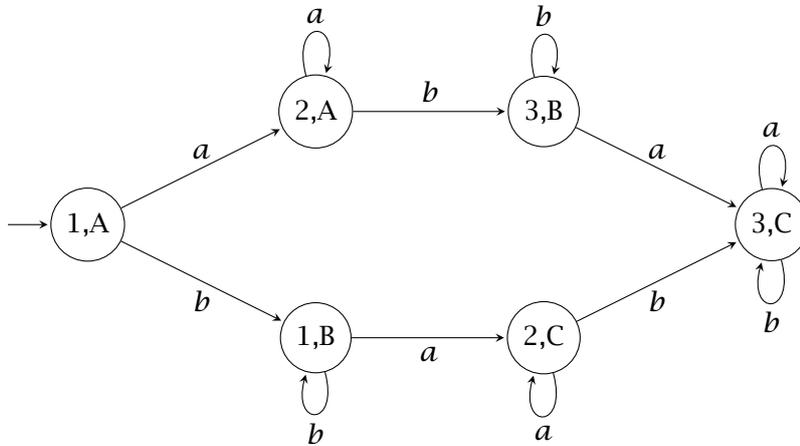


Le produit des machines associées est



Dans le cas de deux automates $Q_1 = (\mathcal{A}, S_1, \delta_1, s_{1,0}, T_1)$ et $Q_2 = (\mathcal{A}, S_2, \delta_2, s_{2,0}, T_2)$ on peut associer simplement un état initial : $(s_{1,0}, s_{2,0})$.

Dans l'exemple ci-dessus l'élagage depuis (1, A) permet d'omettre les états (1, C), (2, B) et (3, A).



On remarque qu'un mot u est reconnu par Q_1 (resp. Q_2) si et seulement si $(s_{1,0}, s_{2,0}) \cdot u \in T_1 \times S_2$ (resp. $(s_{1,0}, s_{2,0}) \cdot u \in S_1 \times T_2$). Ainsi on a le théorème

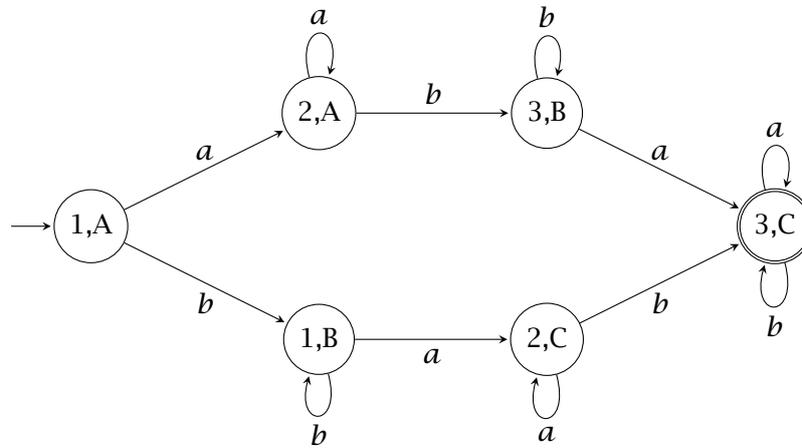
Intersection et union

L'intersection de deux langages reconnaissables est reconnaissable.
 L'union de deux langages reconnaissables est reconnaissable.

Démonstration

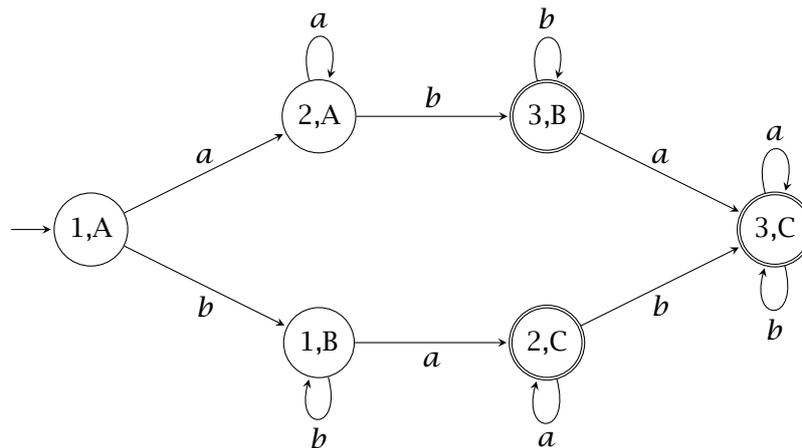
Si L_1 est reconnu par $(\mathcal{A}, S_1, \delta_1, s_{1,0}, T_1)$ et L_2 par $(\mathcal{A}, S_2, \delta_2, s_{2,0}, T_2)$ alors
 $L_1 \cup L_2$ est reconnu par $(\mathcal{A}, S_1 \times S_2, \delta_1 \times \delta_2, (s_{1,0}, s_{2,0}), T_1 \times S_2 \cup S_1 \times T_2)$ et
 $L_1 \cap L_2$ est reconnu par $(\mathcal{A}, S_1 \times S_2, \delta_1 \times \delta_2, (s_{1,0}, s_{2,0}), T_1 \times T_2)$.

Exemple Dans l'exemple ci-dessus, le langage $(L_1 \cap L_2)$ des mots qui contiennent ab et ba est donc reconnu par



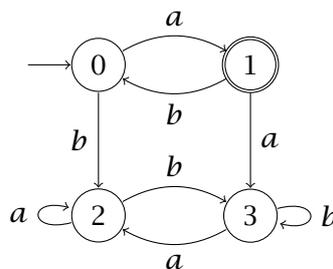
C'est aussi l'ensemble des mots commençant par $b^m a^n b$ ou $a^m b^n a$ avec $m \geq 1$ et $n \geq 1$.

$L_1 \cup L_2$ est reconnu par l'automate



1-5 Automates déterministes incomplets

Nous avons vu qu'on pouvait enlever les états invisibles depuis l'état initial. Il y a d'autres états qu'on peut souhaiter ignorer.



Les états 2 et 3 représentent une sorte de piège : dès qu'on y parvient on ne pourra jamais revenir à un état terminal. La tentation de les enlever se heurte au fait qu'alors il existerait des transitions manquantes.

Automate déterministe incomplet

Un automate fini déterministe incomplet est un quintuplet $Q = (\mathcal{A}, S, \delta, s_0, T)$

- ↪ où \mathcal{A} est un alphabet (fini),
- ↪ S est un ensemble fini d'états,
- ↪ δ est une application définie sur une **partie** de $S \times \mathcal{A}$ vers S : la fonction de transition.
- ↪ s_0 est un élément de S , l'état initial,
- ↪ T est une partie de S , l'ensemble des états finaux,

Mots bloquants

$Q = (\mathcal{A}, S, \delta, s_0, T)$ est un automate déterministe incomplet.

$u = u_1u_2 \dots u_p$ est un mot sur \mathcal{A} .

Si $s_0.u_1 = s_1, s_1.u_2 = s_2, \dots, s_{k-1}.u_k = s_k$ sont définis mais $\delta(s_k, u_{k+1})$ n'est pas défini (avec $k < p$) on dit que u est un **mot bloquant**.

Le langage reconnu par un automate incomplet est l'ensemble des mots u non bloquants tels que $s_0.u$ appartient à T .

Co-accessibilité

$Q = (\mathcal{A}, S, \delta, s_0, T)$ est un automate déterministe.

Un état $s \in S$ est dit co-accessible s'il existe $u \in \mathcal{A}^*$ non bloquant tel que $s.u$ appartient à T .

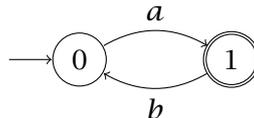
Si S' est l'ensemble des états co-accessibles de Q alors

$Q' = (\mathcal{A}, S', \delta, s_0, T \cap S')$ est un automate déterministe incomplet tel que $L(Q) = L(Q')$.

Démonstration ; tout mot reconnu par Q' est reconnu par Q .

De plus tout mot reconnu par Q est non bloquant et ne passe que par des états co-accessibles donc appartenant à S' , il est donc reconnu par Q' .

Dans l'exemple ci-dessus on aboutit ainsi à l'automate



Comme on a élargi la définition des automates on peut imaginer que l'on a aussi augmenté le nombre de langages reconnaissables. En fait il n'en est rien.

L'idée est d'ajouter un état-puits qui va recevoir toutes les transitions manquantes et qui reste stable par toutes les lettres. On construit ainsi un automate complet.

Complétion

$Q = (\mathcal{A}, S, \delta, s_0, T)$ est un automate incomplet.

On choisit p n'appartenant pas à S et on définit $S' = S \cup \{p\}$.

On prolonge δ en δ' défini sur $S' \times \mathcal{A}$ par

↪ $\delta'(s, x) = \delta(s, x)$ pour $s \in S$ si $\delta(s, x)$ existe,

↪ $\delta'(s, x) = p$ pour $s \in S$ si $\delta(s, x)$ n'est pas défini et

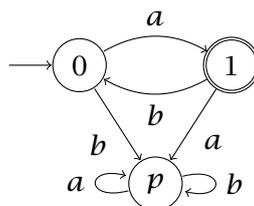
↪ $\delta'(p, x) = p$

$Q' = (\mathcal{A}, S', \delta', s_0, T)$ est un automate déterministe complet tel que

$L(Q) = L(Q')$.

Démonstration : les mots bloquants deviennent alors des mots qui arrivent à l'état-puits et qui y restent donc ne sont pas reconnus alors que les mots non-bloquants gardent leur parcours et sont reconnus ou non en même temps dans les deux automates.

L'exemple ci-dessus donne finalement l'automate



2 Automates non-déterministes

Les automates vus jusqu'à présent envoyait un état vers un autre état bien déterminé (ou vers aucun état) sous l'action d'une lettre.

Nous allons maintenant lever cette contrainte d'un état unique et considérer qu'une lettre peut déterminer plusieurs transitions depuis un état.

Ce non-déterminisme sera à l'œuvre aussi au départ : on autorisera plusieurs états initiaux.

2-1 Définitions

Automate fini non-déterministe

Un automate fini non-déterministe ou AFND (NFA en anglais) est un quintuplet $(\mathcal{A}, S, \Delta, I, T)$ composé de :

- ↪ \mathcal{A} , un alphabet,
- ↪ S , l'ensemble fini des états de l'automate,
- ↪ Δ une application de $S \times \mathcal{A}$ dans $\mathcal{P}(S)$, c'est la fonction de transition.
- ↪ $I \subset S$, l'ensemble des états initiaux,
- ↪ $T \subset S$, l'ensemble des états acceptants,

Un automate déterministe peut ainsi être considéré comme un cas particulier d'automate non-déterministe. Dans le cas d'un automate déterministe, pour tout couple $(s, x) \in S \times \mathcal{A}$, $\Delta(s, x)$ admet exactement un élément dans le cas des automates complets ou au plus un élément dans le cas des automates incomplets.

On peut remplacer la fonction de transition par son graphe $G_\Delta = \{(s, x, s') ; s' \in \Delta(s, x)\}$.

G_Δ est une partie de $S \times \mathcal{A} \times S$, c'est l'ensemble des transitions.

Δ est alors défini par $\Delta(s, x) = (\{s\} \times \{x\} \times S) \cap G_\Delta = \{s' \in S ; (s, x, s') \in G_\Delta\}$.

On notera encore la transition $(s, x, s') \in G_\Delta$ par $s \xrightarrow{x} s'$.

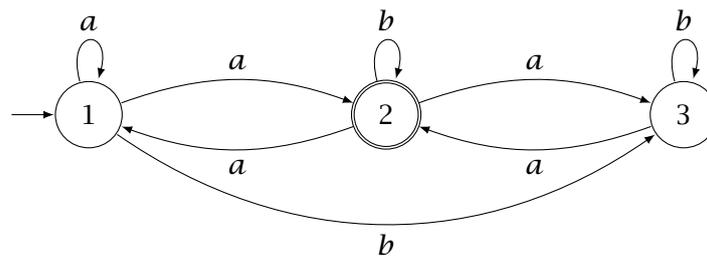
Calcul

- Un calcul de l'automate est une suite (finie) de transitions telle que l'origine de chacune est l'extrémité de la précédente.
- Un calcul sera noté $s \xrightarrow{x_1} s_1 \xrightarrow{x_2} s_2 \xrightarrow{x_3} \dots \xrightarrow{x_{n-1}} s_{n-1} \xrightarrow{x_n} s_n$.
 - s est l'origine du calcul,
 - s_n est l'extrémité du calcul,
 - $u = x_1x_2 \dots x_n$ est l'étiquette du calcul.

La principale différence avec les automates déterministe est qu'il y avait une bijection entre les parcours et leur étiquette.

Dans le cas d'un automate non-déterministe chaque calcul a une étiquette unique mais un même mot peut être associé à plusieurs calculs depuis une même origine.

Exemple



$1 \xrightarrow{a} 1 \xrightarrow{a} 1 \xrightarrow{b} 3,$ $1 \xrightarrow{a} 1 \xrightarrow{a} 2 \xrightarrow{b} 2,$ $1 \xrightarrow{a} 2 \xrightarrow{a} 1 \xrightarrow{b} 3,$ $1 \xrightarrow{a} 2 \xrightarrow{a} 3 \xrightarrow{b} 3$
 sont les calculs d'origine 1 et d'étiquette aab .

Langage associé

- Un calcul est **réussi** si son origine appartient à I et son extrémité appartient à T .
- Un mot est **reconnu** (ou accepté) par l'automate s'il est l'étiquette d'au moins un calcul réussi.
- Le **langage reconnu** (ou langage accepté) par un automate M est l'ensemble, $L(M)$, des mots reconnus par l'automate.

2-2 Déterminisation

Comme les automates déterministes sont des cas particuliers d'automates non-déterministes tout langage reconnaissable est reconnu par un automate non-déterministe.

Par contre il n'est pas évident qu'un langage reconnu par un automate non-déterministe soit reconnaissable (par un automate déterministe). C'est pourtant le cas ; la démonstration s'appuie sur la transformation d'un automate non-déterministe en un automate déterministe en conservant le langage.

Automate des parties

$Q = (\mathcal{A}, S, \Delta, I, T)$ est un automate non-déterministe.

- L'automate des parties associé à Q est $Q' = (\mathcal{A}, \mathcal{P}(S), \delta, I, \mathcal{F})$ avec
 - $\rightarrow \delta(E, x) = \bigcup_{s \in E} \Delta(s, x) = \{s' \in S ; \exists s \in E, (s, x, s') \in G_\Delta\}$.
 - $\rightarrow \mathcal{F} = \{E \subset S ; E \cap T \neq \emptyset\}$
- Q' est un automate déterministe complet.
- $L(Q') = L(Q)$.

On dit aussi que Q' est le déterminisé de Q .

Démonstration

1. I est une partie de S donc $I \in \mathcal{P}(S)$.

De même $\delta(E, x) \in \mathcal{P}(S)$ pour tout $E \in \mathcal{P}(S)$ et pour tout $x \in \mathcal{A}$.
 Q' est bien un automate déterministe complet.

2. Si $u = x_1 x_2 \dots x_n$ appartient à $L(Q)$ alors il existe un calcul réussi

$s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} s_2 \xrightarrow{x_3} \dots \xrightarrow{x_{n-1}} s_{n-1} \xrightarrow{x_n} s_n$ avec $s_0 \in I$ et $s_n \in T$.

On a alors $s_1 \in \Delta(s_0, x_1)$ avec $s_0 \in I$ donc $s_1 \in \delta(I, x_1) = E_1$.

De même $s_2 \in \Delta(s_0, x_1)$ avec $s_1 \in E_1$ d'où $s_2 \in \delta(E_1, x_2) = E_2$.

On construit ainsi le parcours de u dans Q' : $I \xrightarrow{x_1} E_1 \xrightarrow{x_2} E_2 \xrightarrow{x_3} \dots \xrightarrow{x_{n-1}} E_{n-1} \xrightarrow{x_n} E_n$

Or $s_n \in E_n$ et $s_n \in T$ donc $E_n \cap T \neq \emptyset$ d'où $E_n \in \mathcal{T}$.

Ainsi $I.u \in \mathcal{T}$ dans Q' donc $u \in L(Q') : on a prouvé $L(Q) \subset L(Q')$.$

3. Inversement si $u \in L(Q')$ alors son parcours est de la forme

$I = E_0 \xrightarrow{x_1} E_1 \xrightarrow{x_2} \dots \xrightarrow{x_{n-1}} E_{n-1} \xrightarrow{x_n} E_n$ avec $E_n \in \mathcal{T}$ donc il existe $s_n \in E_n \cap T$.

La transition $E_{n-1} \rightarrow E_n$ implique qu'il existe un élément s_{n-1} dans E_{n-1} et une transition $s_{n-1} \xrightarrow{x_n} s_n$

On définit ainsi, en descendant, un élément $s_i \in E_i$ pour tout i avec un calcul

$$s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} \dots \xrightarrow{x_{n-1}} s_{n-1} \xrightarrow{x_n} s_n$$

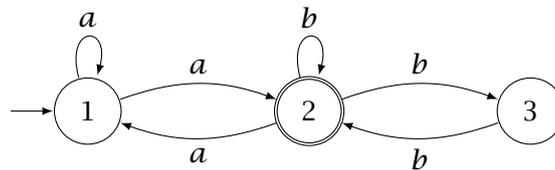
$s_0 \in E_0 = I, s_n \in T$ donc u est reconnu par Q .

On a alors $L(M') \subset L(M)$ d'où l'égalité.

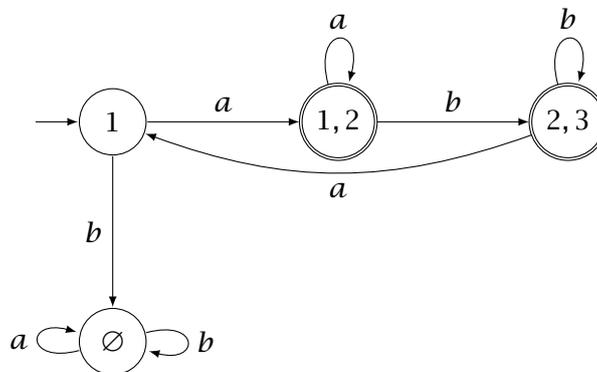
Si l'automate initial est non-déterministe avec n états, le déterminisé admet avec 2^n états.

Dans le calcul pratique on ne calculera les images des parties de S que pour les parties accessibles : on part de I on calcule $\delta(I, x)$ pour les différentes lettres de Σ puis les images de ces derniers ensembles et ainsi de suite : on calcule seulement l'automate émondé.

Exemple



Q'	$\{1\}$	$\{1, 2\}$	$\{2, 3\}$	\emptyset
a	$\{1, 2\}$	$\{1, 2\}$	$\{1\}$	\emptyset
b	\emptyset	$\{2, 3\}$	$\{2, 3\}$	\emptyset



3 L'algorithme de Berry-Sethi

Les langages rationnels et les langages reconnaissables ont des propriétés qui en permettent une implémentation par une machine : les expressions régulières pour les premiers ou les automates pour les seconds. Il serait utile de pouvoir utiliser ces deux outils conjointement.

De fait ces deux dénominations décrivent les mêmes langages : il n'y a pas de différence entre les langages rationnels et les langages reconnaissables.

Nous allons démontrer dans cette partie une implication : tout langage reconnaissable et rationnel. La réciproque est hors-programme.

Nous utiliserons un résultat du chapitre précédent :

Si e est une expression régulière linéaire (c'est-à-dire dont toutes les lettres employées sont distinctes) alors le langage qu'elle dénote est un langage local.

Les démonstrations des résultats sont souvent demandées en exercices.

3-1 Automate local

L est un langage local sur \mathcal{A} défini par (P, F, S) .

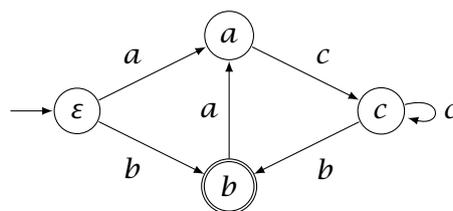
Automate de Glushkov

L'automate de Glushkov associé à L est l'automate incomplet $Q = (\mathcal{A}, \mathcal{A} \cup \{\varepsilon\}, \delta, \varepsilon, S')$ où δ est définie par

- $\hookrightarrow \delta(\varepsilon, x)$ existe si et seulement si $x \in P$, alors $\delta(\varepsilon, x) = x$,
- $\hookrightarrow \delta(x, y)$ existe si et seulement si $xy \in F$, alors $\delta(x, y) = y$.

et $S' = S \cup \{\varepsilon\}$ si $\varepsilon \in L$, $S' = S$ sinon.

Exemple L est le langage local sur $\mathcal{A} = \{a, b, c\}$ défini par $P = \{a, b\}$, $S = \{b\}$, $F = \{ac, ba, cb, cc\}$ et ne contenant pas ε .



On remarquera que toutes les transitions de même étiquette ont la même extrémité.

Théorème de Glushkov

Si Q est l'automate de Glushkov associé à L alors Q reconnaît L .

On peut synthétiser ce résultat

$$\begin{array}{ccc}
 L_r & \xleftarrow{=} & L(Q) \\
 \uparrow & & \uparrow \\
 r & \longrightarrow & Q
 \end{array}$$

3-2 Morphismes

On considère une fonction f d'un alphabet \mathcal{B} vers un alphabet \mathcal{A} .

1. Les mots sont des produits de lettres : on peut donc définir une fonction f^* de \mathcal{B}^* vers \mathcal{A}^* par $f^*(\varepsilon) = \varepsilon$ (le premier ε est le mot vide de \mathcal{B}^* , le second celui de \mathcal{A}^*) et $f^*(u.x) = f^*(u).f(x)$ pour $u \in \mathcal{B}^*$ et $x \in \mathcal{B}$.
On prolonge f^* sur les langages sans changer le nom :
si L est un langage sur \mathcal{B} , $f^*(L) = \{f^*(u) ; u \in L\}$ est un langage sur \mathcal{A} .
2. De même on peut prolonger f de manière inductive sur l'ensemble des expressions régulières sur \mathcal{B} :
 $f(\mathbf{0}) = \mathbf{0}$, $f(\mathbf{1}) = \mathbf{1}$, $f(\mathbf{a}) = \mathbf{b}$ avec $b = f(a)$,
 $f(r_1 + r_2) = (f(r_1) + f(r_2))$, $f(r_1.r_2) = (f(r_1).f(r_2))$ et $f(r^*) = (f(r))^*$.

Transport des expressions régulières

Si f est une fonction de \mathcal{B} vers \mathcal{A} alors $L[f(\mathbf{r})] = f^*(L[\mathbf{r}])$
pour toute expression régulière \mathbf{r} sur \mathcal{B} .

On a donc un transport depuis les expressions régulières vers les langages.

$$\begin{array}{ccc} L[r] & \xrightarrow{f^*} & L[r1] \\ \uparrow & & \uparrow \\ r & \xrightarrow{f} & r1 \end{array}$$

Dans ce diagramme les deux chemins de r à $L[r1]$ donnent le même résultat.

3. f peut servir aussi à transformer les automates (non déterministes) sur l'alphabet \mathcal{B} . Si $Q = (\mathcal{B}, S, \Delta, I, T)$ on note $\hat{f}(Q) = (\mathcal{A}, S, \Delta', I, T)$ où le graphe de Δ' est défini par $G_{\Delta'} = \{(s, f(x), s') ; (s, x, s') \in G_{\Delta}\}$. C'est l'automate obtenu en changeant les lettres par leur image par f .

Transport des automates

Si f est une fonction de \mathcal{B} vers \mathcal{A} alors $L(\hat{f}(Q)) = f^*(L(Q))$.
pour toute expression régulière r sur \mathcal{B} .

On a donc un transport depuis les automates vers les langages.

$$\begin{array}{ccc} L(Q) & \xrightarrow{f^*} & L(Q') \\ \uparrow & & \uparrow \\ Q & \xrightarrow{\hat{f}} & Q' \end{array}$$

Dans ce diagramme les deux chemins de Q à $L(Q')$ donnent le même résultat.

3-3 Construction de l'automate

1. On part d'un langage rationnel non vide L .
dénové par une expression régulière e , $L = L[e]$.
2. On transforme e en une expression linéaire $e1$ en changeant uniquement les lettres par des lettres **distinctes**.

Si on note \mathcal{A}_1 l'alphabet utilisé dans $e1$, on note f la fonction de \mathcal{A}_1 vers \mathcal{A} qui donne, pour chaque lettre de \mathcal{A}_1 , la lettre de \mathcal{A} qu'elle remplace.

Avec la notation ci-dessus on a $e = f(e1)$.

3. On note $L_1 = L[e1]$, L_1 est local car $e1$ est linéaire.
4. On associe à L_1 son automate de Glushkov, Q_1 .
5. On transforme Q_1 en Q en remplaçant les étiquettes à l'aide de $f : Q = \hat{f}(Q_1)$.
6. On peut, si besoin, déterminer Q .

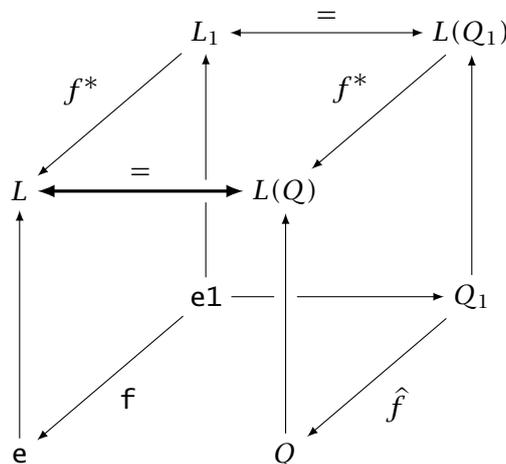
Tout langage rationnel est reconnaissable

L'algorithme de Berry-Sethi construit un automate qui reconnaît le langage initial.

C'est-à-dire $L = L(Q)$.

Il suffit de rassembler les résultats prouvés ci-dessus.

1. $L = L[e] = L[f(e1)] = f^*(L[e1]) = f^*(L_1)$ (transport des expressions régulières)
2. $L_1 = L(Q_1)$ (théorème de Glushkov)
3. $L(Q) = L(\hat{f}(Q_1)) = f^*(L(Q_1))$ (transport des automates)
4. Ainsi $L = f^*(L_1) = f^*(L(Q_1)) = L(Q)$.



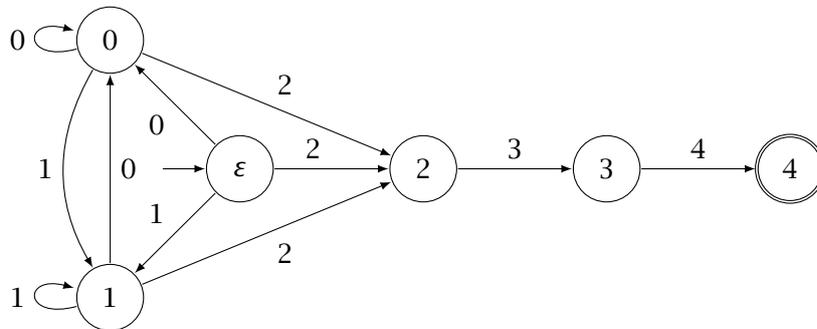
3-4 Un exemple

1. Soit L dénoté par $e = (a+b)^*.a.b.a$.
2. On pose $e1 = (0+1)^*.2.3.4$: on définit donc f par le tableau $[a; b; a; b; a]$.

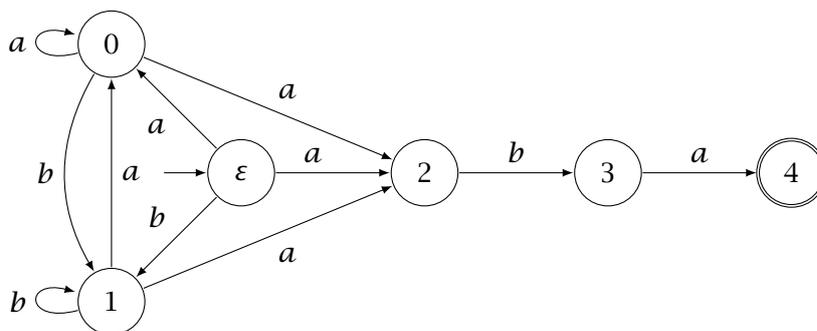
3. $L_1 = L[e1]$ ne contient pas ϵ .

On calcule $P = \{0, 1, 2\}$, $S = \{4\}$, $F = \{00, 01, 10, 11, 02, 12, 23, 34\}$.

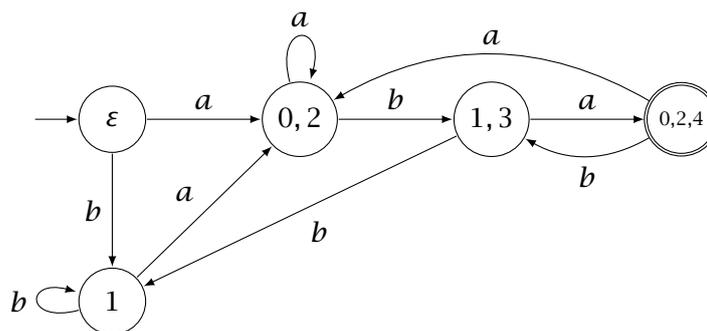
4. L'automate Q_1 est calculé.



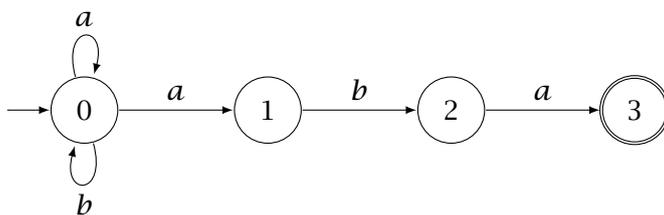
5. On traduit en l'automate Q , non déterministe.



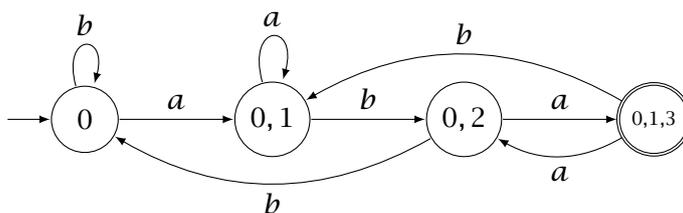
6. On peut le déterminer.



On pouvait aussi trouver directement un automate non déterministe



puis le déterminer



4 Exercices

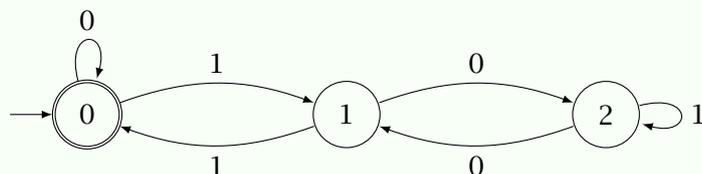
4-1 Automates déterministes

Ex. 1 Produit de 2 mots Prouver que $s.(u.v) = (s.u).v$ pour $u, v \in \mathcal{A}^*$.

Ex. 2 3 occurrences \mathcal{A} désigne l'alphabet $\{a, b\}$ et L l'ensemble des mots qui contiennent au moins 3 occurrences de la lettre a : $L = \{u \in \Sigma^* / |u|_a \geq 3\}$.
Donner une expression rationnelle dénotant L .
Décrire un automate fini reconnaissant L .

Ex. 3 Divisibilité par trois

Prouver que l'automate suivant teste la divisibilité par trois d'un nombre exprimé en binaire.



Que se passe-t-il si on change l'état final ?

Ex. 4 Petits automates

Énumérer tous les automates déterministes complets à 2 états sur $\mathcal{A} = \{a, b\}$ et le langage qu'ils reconnaissent.

Ex. 5 Alphabet à une lettre

Déterminez la forme des automates déterministes sur un alphabet à une lettre.
En déduire la structure des langages reconnaissables sur un alphabet à une lettre.

Ex. 6 États co-accessibles À quelle condition sur le langage reconnu un automate complet n'a-t-il que des états co-accessibles ?

4-2 Automates non déterministes

Ex. 7 Cas du mot vide Dans la démonstration de l'égalité des langages d'un automate non déterministe et de son déterminisé, on n'a pas considéré le cas du mot vide. Compléter la démonstration.

Ex. 8 Construction

Construire un automate reconnaissant tous les mots qui finissent par aba . Déterminiser l'automate obtenu.

Ex. 9 Le pire peut arriver Donner un automate (non-déterministe) à $n + 1$ états qui reconnaît L , le langage sur $\mathcal{A} = \{a, b\}$ des mots de n lettres au moins qui finissent par $a.u$ avec u de longueur $n - 1$. Prouver que tout automate déterministe qui reconnaît L admet au moins 2^n états.

Ex. 10 Langage transposé

$Q = (\mathcal{A}, S, \Delta, I, T)$ est un automate non déterministe.

On définit $Q^T = (\mathcal{A}, S, \Delta^T, T, I)$ avec Δ^T tel que $G_{\Delta^T} = \{(s, x, s') ; (s', x, s) \in G_{\Delta}\}$.

Q^T est l'automate obtenu en inversant les transitions.

Pour tout mot $u = u_1 u_2 \dots u_n$ le miroir de u est $u^T = u_n u_{n-1} \dots u_1$.

Pour tout langage L le langage miroir de L , L^T , est $L^T = \{u^T ; u \in L\}$.

Prouver que $L(Q^T) = (L(Q))^T$.

Ex. 11 Le barman aveugle

On dispose de 4 jetons, chacun ayant une face noire et une face blanche. Un joueur (le barman) a les yeux bandés. Son but est de retourner les 4 jetons sur la même couleur (dès que les 4 jetons sont retournés la partie s'arrête et le barman a gagné). Pour cela, il peut retourner à chaque tour 1, 2 ou 3 jetons. Un autre joueur perturbe le jeu en tournant le plateau sur lequel reposent les jetons d'un quart de tour, d'un demi-tour ou de trois quarts de tour entre chaque opération du barman. Montrer que le barman a une stratégie gagnante, c'est-à-dire que quoi que fasse celui qui tourne le plateau, le barman gagnera.

4-3 Reconnaissabilité des langages rationnels

Ex. 12 Théorème de Glushkov Prouver que si Q est l'automate de Glushkov associé à un langage local L alors $L = L(Q)$.

Ex. 13 Langage reconnu par un automate local

Un automate déterministe $Q = (\mathcal{A}, S, \delta, s_0, T)$ est local si, pour tout $x \in \mathcal{A}$, $\delta(s, x)$ est indépendant de s .

Prouver que le langage reconnu par un automate local est un langage local.

Ex. 14 Morphisme Prouver que si f est une fonction de \mathcal{B} vers \mathcal{A} alors

$f^*(u.v) = f^*(u).f^*(v)$ pour tous mots u et v de \mathcal{B}^* .

Prouver que si L_1 et L_2 sont des langages sur \mathcal{B} alors

$f^*(L_1 \cup L_2) = f^*(L_1) \cup f^*(L_2)$, $f^*(L_1.L_2) = f^*(L_1).f^*(L_2)$ et $f^*(L_1^*) = (f^*(L_1))^*$.

A-t-on toujours $f^*(L_1 \cap L_2) = f^*(L_1) \cap f^*(L_2)$?

Ex. 15 Transport des expressions régulières

Prouver que $L[f(x)] = f^*(L[x])$.

Ex. 16 Transport des automates Prouver que $L(\hat{f}(Q)) = f^*(L(Q))$.

Chapitre IX

Langages rationnels compléments

1	Algorithme de Mc Naughton et Yamada	157
2	Lemme de l'étoile (ou de pompage)	159
3	Langages dérivés	161
3-1	Automate dérivé	162
3-2	Le morphisme canonique	163
4	Minimisation	164

Les résultats qui suivent ne sont pas explicitement au programme. Ils permettent d'approfondir la connaissance des langages rationnels/reconnaissables

1. *On commence par prouver, en construisant une expression régulière, qu'un langage reconnaissable est rationnel.
Ce résultat est la réciproque du théorème de Kleene ; il permet d'employer indistinctement rationnel et reconnaissable, ce qui est fait ici.
On notera que les expressions régulières construites ne sont en général pas optimales, il ne sera pas possible de les utiliser humainement même pour les automates simples.*
2. *On donne un critère utilisable pour prouver qu'un langage n'est pas rationnel. Ce résultat montre aussi que les automates qui reconnaissent des langages finis sont "gros".*
3. *On donne une caractérisation algébrique des langages rationnels.
On construit alors un automate canoniquement associé à un langage rationnel.
On prouve que cet automate est de taille minimale et que tout automate de taille minimale lui est isomorphe.*
4. *On donne enfin une méthode algorithmique pour construire effectivement cet automate minimal.*

1 Algorithme de Mc Naughton et Yamada

$Q = (\mathcal{A}, S, \Delta, I, T)$ est un automate non déterministe. On rappelle qu'un automate déterministe peut être considéré comme un automate déterministe.

On veut prouver que le langage reconnu par Q est rationnel.

- Le langage reconnu est l'ensemble des mots pour lesquels il existe un calcul réussi entre un état de I et un état de T : c'est donc l'union des langages reconnus par $(\mathcal{A}, S, \Delta, \{s\}, \{t\})$ pour $s \in S$ et $t \in T$. On va prouver le résultat dans le cas où les ensembles des états finaux et des états initiaux sont des singleton ; on obtiendra un expression rationnelle en faisant la somme des expressions rationnelles construites.
- Pour 2 états s et t dans S , on note $L(s, t)$ est le langage reconnu par $(\mathcal{A}, S, \Delta, \{s\}, \{t\})$.
- On sait facilement calculer les chemins simples, sans intermédiaires entre deux sommets états s et t : c'est l'ensemble des lettres x telles que (s, x, t) est une transition auquel on ajoute ϵ si $s = t$.
- L'idée de l'algorithme de Mc Naughton et Yamada (MNY) est de calculer les langages pas-à-pas en ajoutant à chaque étape un état supplémentaire par lequel les calculs peuvent passer.

On supposera que les états sont décrits par les entier de 0 à $N - 1$, où N est la taille de l'automate.

Langages partiels d'ordre k

Si $Q = (\mathcal{A}, S, \Delta, I, T)$ est un automate tel que $S = \{0, 1, 2, \dots, N - 1\}$
 $L^k(s, t)$ est l'ensemble des étiquettes des calculs entre s et t
 ne passant que par des états de $\{0, 1, 2, \dots, k - 1\}$
 en dehors de l'origine et de l'extrémité.

- ↪ On a vu ci-dessus montre que l'on sait calculer facilement les $L^0(s, t)$.
- ↪ On a donc un algorithme qui applique la programmation dynamique si on sait calculer les $L^{k+1}(s, t)$ en fonctions des $L^k(u, v)$.
- ↪ On aura alors $L(s, t) = L^N(s, t)$.

Le principal outil est de remarquer que si u appartient à $L^{k+1}(s, t)$ alors

↪ soit u appartient à $L^k(s, t)$

↪ soit le calcul est composé d'un calcul de s à k , de calculs de k à k puis d'un calcul de k à t , chacun de ces calculs ne passant que par des états de $\{1, 2, \dots, k\}$.

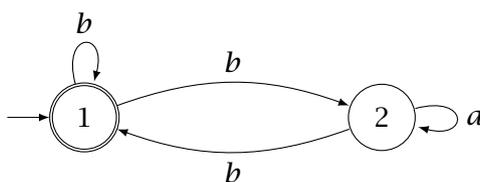
$$L^{k+1}(s, t) = L^k(s, t) \cup L^k(s, k).(L^k(k, k))^*.L^k(k, t)$$

Comme les langages $L^k(s, t)$ sont finis donc rationnels et que l'on n'effectue que des opérations rationnelles (union, produit, étoile) on en déduit que les langages $L(s, t) = L^N(s, t)$ sont rationnels puis que $L(Q) = \bigcup_{s \in I, t \in F} L(s, t)$ est rationnel.

Rationalité des langages reconnaissables

Le langage reconnu par un automate est rationnel.

Exemple :



On calculera les expressions régulières avec des règles de priorité.

On commence par la matrice des $L^0(s, t)$: $A_0 = \begin{pmatrix} b & b \\ b & a \end{pmatrix}$.

On applique l'itération une première fois $A_1 = \begin{pmatrix} b + b.b * .b & b + b.b * .b \\ b + b.b * .b & a + b.b * .b \end{pmatrix}$.

On se contente de calculer une expression régulière dénotant $L^2(1, 1)$:

$$1 + b + b.b * .b + (b + b.b * .b).(a + b.b * .b) * .(b + b.b * .b)$$

Ce n'est pas l'expression régulière la plus simple : une expression plus simple dénotant le langage est $(b + b.a * .b)*$.

Ex. 1 Complexité

Quelle est l'ordre de grandeur du nombre d'opérations nécessaires à l'algorithme de Mc Naughton et Yamada ?

2 Lemme de l'étoile (ou de pompage)

Il est, pour l'instant, difficile de prouver qu'un langage n'est pas reconnaissable ou rationnel.

Il faudrait prouver qu'il n'existe aucune expression régulière le dénotant ou aucun automate le reconnaissant : un tel résultat négatif semble hors de portée.

Un moyen classique pour nier une propriété est de tirer une conséquence de celle-ci qui est plus facilement contredite ; nous allons ici montrer une telle propriété. Le résultat, assez technique, n'a pas à être connu mais la démonstration est assez simple pour pouvoir être reproduite si nécessaire.

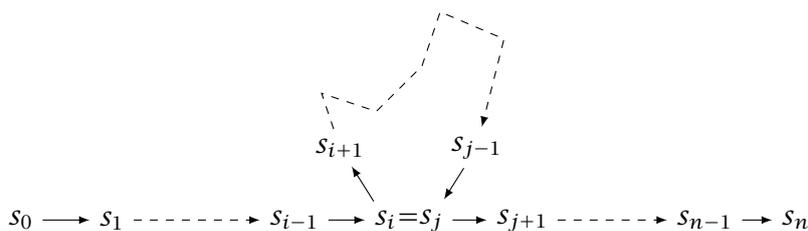
On suppose que le langage L est reconnu par l'automate (non déterministe) $Q = (\mathcal{A}, S, \Delta, I, T)$.

Le cardinal de S , le nombre d'états, est appelé aussi taille de Q , on le note N .

Pour tout mot u de L de longueur $n \geq N$ on considère un calcul réussi de u dans Q :

$$s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} \dots \xrightarrow{x_{N-1}} s_{N-1} \xrightarrow{x_N} s_N \xrightarrow{x_{N+1}} \dots \xrightarrow{x_{n-1}} s_{n-1} \xrightarrow{x_n} s_n$$

L'ensemble de $N + 1$ éléments $\{s_0, s_1, \dots, s_N\}$ est inclus dans S de cardinal N donc il existe deux indices i et j distincts tels que $0 \leq i < j \leq N$ et $s_i = s_j$.



Si on définit $u_1 = x_1 \dots x_i$, $u_2 = x_{i+1} \dots x_j$ et $u_3 = x_{j+1} \dots x_n$ on a

- ↪ $u = u_1.u_2.u_3$,
- ↪ $|u_1.u_2| = j \leq N$,
- ↪ $|u_2| = j - i \geq 1$,
- ↪ $s_0.u_1 = s_i$
- ↪ $s_i.u_2 = (s_0.u_1).u_2 = s_0.(u_1.u_2) = s_j = s_i$.
- ↪ $s_i.u_3 = s_j.u_3 = (s_0.(u_1.u_2)).u_3 = s_0.u = s_n \in T$.

On en déduit, par récurrence sur k , que $s_i.u_2^k = s_i$ pour tout $k \in \mathbb{N}$ d'où

$$s_0.(u_1.u_2^k.u_3) = ((s_0.u_1).u_2^k).u_3 = (s_i.u_2^k).u_3 = s_i.u_3 = s_n \in T$$

Ainsi tous les mots $u_1.u_2^k.u_3$ sont reconnus.

Lemme de l'étoile ou Pumping lemma

Si L est un langage reconnaissable alors il existe un entier N tel que tout mot $u \in L$ de longueur supérieure à N peut se décomposer en $u = u_1 u_2 u_3$ avec $|u_1 u_2| \leq N$, $|u_2| \geq 1$ et $\forall k \in \mathbb{N}$, $u_1 u_2^k u_3 \in L$.

Exemples

- Soit $L = \{a^n b^n ; n \in \mathbb{N}\}$.
 Si L était reconnaissable on pourrait, pour l'entier N du lemme de l'étoile, décomposer $u = a^N b^N \in L$ en $u_1.u_2.u_3$ avec $|u_1.u_2| \leq N$ et $|u_2| \geq 1$.
 On en déduit que $u_1 = a^p$, $u_2 = a^q$ avec $q \geq 1$ et $u_3 = a^r b^N$ avec $p + q + r = N$.
 On devrait avoir $u_1 u_2^k u_3 = a^{N+(k-1)q} b^N \in L$ ce qui est impossible pour $k \neq 1$: L n'est pas reconnaissable.
- On peut en déduire que $L' = \{u \in \{a, b\}^* ; |u|_a = |u|_b\}$ n'est pas rationnel.
 En effet son intersection avec $L_{a^* b^*}$ est L donc la rationalité de L' impliquerait celle de L .
- Un mot de **Dyck** est un mot $u \in \{a, b\}^*$ tel que $\begin{cases} |u|_a = |u|_b \\ \text{si } u = v.w, \text{ alors } |v|_a \geq |v|_b \end{cases}$
 C'est la formalisation des parenthésages : si a représente "(" et b représente ")" les mots de Dyck sont les parenthésages corrects.
 On note D l'ensemble des mots de Dyck.
 Si D était reconnaissable on pourrait, pour l'entier N du lemme de l'étoile, décomposer $u = a^N b^N \in D$ en $u_1.u_2.u_3$ avec $|u_1.u_2| \leq N$ et $|u_2| \geq 1$.
 On en déduit que $u_1 = a^p$, $u_2 = a^q$ avec $q \geq 1$ et $u_3 = a^r b^N$ avec $p + q + r = N$.
 On devrait avoir $u_1.u_2^k.u_3 = a^{N+(k-1)q} b^N \in D$ ce qui est impossible pour $k \neq 1$: D n'est pas reconnaissable.
- Si L est rationnel et s'il existe au moins un mot auquel on peut appliquer la décomposition alors le langage contient $\{u_1.u_2^n.u_3 ; n \in \mathbb{N}\}$ avec $u_2 \neq \varepsilon$; tous ces mots sont distincts donc le langage est infini.
- Inversement si le langage L est fini (il est alors rationnel) on ne peut appliquer la décomposition à aucun mot. On en déduit qu'aucun mot n'est de longueur supérieure à N .
 Ainsi un automate qui reconnaît un langage fini a un nombre d'états strictement supérieur à la longueur du plus long mot de L .

3 Langages dérivés

Langage dérivé

Si L est un langage sur \mathcal{A} et si u est un mot ($u \in \mathcal{A}^*$) le langage dérivé (à gauche) de L par u est $u^{-1}.L = \{v \in \mathcal{A}^* / u.v \in L\}$.
On dit aussi que $u^{-1}.L$ est un **résiduel** de L .

Premières propriétés

1. ε appartient à $u^{-1}.L$ si et seulement si $u \in L$.
2. $\varepsilon^{-1}.L = L$.
3. $u^{-1}.\mathcal{A}^* = \mathcal{A}^*$.
4. $v^{-1}.(u^{-1}.L) = (u.v)^{-1}.L$.

Ex. 2 Démontrer ces propriétés.

Ex. 3 Un exemple

On note L_1 le langage sur $\mathcal{A} = \{a, b\}$ des mots ayant un nombre pair de b et L_2 le langage des mots ayant un nombre impair de b .

Calculer les dérivés $u^{-1}.L_1$ et $u^{-1}.L_2$ pour un mot de \mathcal{A}^* .

Ex. 4 Un autre exemple

On pose $L = \{a^n b^n ; n \in \mathbb{N}\}$ sur $\mathcal{A} = \{a, b\}$.

Prouver que $(a^p)^{-1}.L = \{a^n b^{n+p} ; n \in \mathbb{N}\}$ pour $p \geq 1$.

Prouver que $(a^p.b^q)^{-1}.L = \{b^{p-q}\}$ pour $p \geq q \geq 1$.

Ex. 5 Autres propriétés

L_1 et L_2 sont des langages sur \mathcal{A} et a est une lettre de \mathcal{A} .

1. Prouver que $a^{-1}(L_1 \cup L_2) = a^{-1}L_1 \cup a^{-1}L_2$.
2. Prouver que si $\varepsilon \notin L_1$ alors $a^{-1}.(L_1.L_2) = (a^{-1}.L_1).L_2$.
3. Prouver que si $\varepsilon \in L_1$ alors $a^{-1}.(L_1.L_2) = a^{-1}.L_2 \cup (a^{-1}.L_1).L_2$.
4. Prouver que $a^{-1}.(L_1^*) = (a^{-1}.L_1).L_1^*$.

3-1 Automate dérivé

L est un langage reconnaissable.

$Q = (\mathcal{A}, S, \delta, s_0, T)$ est un automate déterministe complet reconnaissant L .

$v \in u^{-1}.L \iff uv \in L \iff s_0.(uv) \in T \iff (s_0.u).v \in T$.

Ainsi $u^{-1}.L$ est le langage reconnu par $Q_u = (\mathcal{A}, S, \delta, s_0.u, T)$.

On a ainsi une autre démonstration de la rationalité de $u^{-1}.L$.

Q_u est un des automates de la forme $(\mathcal{A}, S, \delta, s.u, T)$ où s est un élément de S : il n'y a qu'un nombre fini de tels automates.

On a ainsi prouvé qu'un langage reconnaissable a un nombre fini de résiduels.

Comme le lemme de l'étoile ce critère peut être utilisé pour prouver la non-reconnaissabilité.

$L = \{a^n b^n ; n \in \mathbb{N}\}$ admet une infinité de dérivés, par exemple les singletons $\{b^r\}$, donc il ne peut pas être reconnaissable.

Inversement si un langage L sur l'alphabet \mathcal{A} admet un nombre fini de langages dérivés on définit un automate, l'**automate dérivé**, $Q_L = (\mathcal{A}, \Lambda, \delta_L, L, \Lambda_T)$ où

$\hookrightarrow \Lambda$ est l'ensemble des résiduels de L , il contient $L = \varepsilon^{-1}.L$,

$\hookrightarrow \Lambda_T$ est l'ensemble des résiduels contenant ε ,

$\hookrightarrow \delta_L(\lambda, x) = x^{-1}.\lambda$ pour $x \in \mathcal{A}$ et pour tout résiduel λ .

Par récurrence sur $|u|$ on montre que $\lambda.u = u^{-1}.\lambda$ pour tout λ .

En particulier $L.u = u^{-1}.L$ donc les mots reconnus sont les mots tels que $u^{-1}.L \in \Lambda_0$ c'est-à-dire les mots u tels que $\varepsilon \in u^{-1}.L$. On a vu que cela caractérisait les mots de L donc L est le langage reconnu par Q .

On a ainsi une caractérisation des langages reconnaissables :

Critère de rationalité

Un langage est rationnel si et seulement si il admet un nombre fini de résiduels.

3-2 Le morphisme canonique

Dans cette partie on considère un langage rationnel L .

L'ensemble fini de ses résiduels est $\Lambda = \{u^{-1}.L ; u \in \mathcal{A}^*\}$.

Soit $Q = (\mathcal{A}, S, \delta, s_0, T)$ un automate déterministe émondé complet qui reconnaît L .

On note $L(s)$ le langage reconnu par $(\mathcal{A}, S, \delta, s, T)$ pour tout sommet de Q .

Si s est un sommet de Q , il est accessible donc il existe $u \in \mathcal{A}^*$ tel que $s = s_0.u$. On a vu qu'alors le langage reconnu on a $L(s) = u^{-1}.L$.

Ainsi l'application qui à $s \mapsto L(s)$ est à valeur dans Λ .

Cette application est surjective car $u^{-1}.L = L(s_0.u)$.

Ainsi le cardinal de S est supérieur ou égal au cardinal de Λ .

L'automate dérivé est donc minimal parmi les automates déterministes complets reconnaissant L .

Si $s = s_0.u$ et si a est une lettre alors

$$L(\delta(s, a)) = L((s_0.u).a) = L(s_0.(u.a)) = (u.a)^{-1}.L = a^{-1}.(u^{-1}.L) = \delta_L(u^{-1}.L, a).$$

Ainsi l'application $s \mapsto L(s)$ "transporte" les actions des automates.

Comme, de plus, $L(s_0) = L$ c'est un morphisme d'automates.

Morphismes d'automates

Un morphisme entre deux automates sur un même langage \mathcal{A} ,

$$Q = (\mathcal{A}, S, \delta, s_0, T) \text{ et } Q' = (\mathcal{A}, S', \delta', s'_0, T')$$

, sont équivalents

s'il existe une bijection p de S vers S' telle que

1. $p(s_0) = s'_0$,
 2. $p(T) = T'$ et
 3. $\delta'(p(s), x) = p(\delta(s, x))$ pour tout $s \in S$ et pour tout $x \in \mathcal{A}$.
- p est un **isomorphisme** de Q vers Q' .

Automates équivalents

Deux automates sur un même langage \mathcal{A} , $Q = (\mathcal{A}, S, \delta, s_0, T)$ et $Q' = (\mathcal{A}, S', \delta', s'_0, T')$, sont équivalents s'il existe une bijection p de S vers S' telle que

1. $p(s_0) = s'_0$,
2. $p(T) = T'$ et
3. $\delta'(p(s), x) = p(\delta(s, x))$ pour tout $s \in S$ et pour tout $x \in \mathcal{A}$.

p est un **isomorphisme** de Q vers Q' .

Ex. 6 Automates minimums

Prouver que si $|S| = N$ alors Q est équivalent à l'automate dérivé.

4 Minimisation

L'étude des résiduels ci-dessus donne un moyen théorique pour calculer un automate minimum (unique à équivalence près) à partir d'un langage.

Cependant il est nécessaire de calculer l'ensemble des langages dérivés et il ne semble pas y avoir d'algorithme raisonnable pour le faire ; la difficulté étant de reconnaître si deux résiduels sont égaux.

De plus un langage rationnel sera souvent connu par un automate le reconnaissant ; il semble plus utile d'essayer de construire un automate minimal à partir d'un automate donné.

C'est un problème important et il existe de nombreux algorithmes pour le faire⁴.

On se donne un automate **déterministe** $Q = (\mathcal{A}, S, \delta, s_0, F)$ qui reconnaît le langage L .

- On veut déterminer un automate déterministe complet et émondé de taille minimale reconnaissant le même langage.
- On rappelle qu'il est unique à équivalence près.

⁴ On peut se référer au document assez exhaustif suivant :
<http://www.di.ens.fr/~jv/HomePage/dea/MinimizeAutomata.pdf>

- Pour chaque état s on note $Q_t = (\mathcal{A}, S, \delta, s, T)$ l'automate issu de Q en remplaçant l'état initial par s et $L_s = L(Q_s)$, le langage reconnu par Q_s .
- On a vu que, si $s = s_0.u$, $L(Q_s) = u^{-1}.L : L_s$ est un résiduel de L .

Équivalence de sommets

On dit que deux états s et t sont équivalents si $L_s = L_t$: on note $s \equiv t$.

Ex. 7 Propriété de l'équivalence

1. Prouver que \equiv définit une relation d'équivalence.
2. Prouver que $s \in F$ et $s \equiv t$ impliquent $t \in F$.
3. Prouver que $s \equiv t$ implique $s.x \equiv t.x$ pour tout $x \in \mathcal{A}$.

- ↪ On note $\sigma(s)$ la classe d'équivalence de s , c'est l'ensemble des états équivalents à s .
- ↪ σ_0 est la classe d'équivalence de s_0 .
- ↪ On note \hat{S} l'ensemble des classe d'équivalences.
- ↪ \hat{T} est l'ensemble des classes d'équivalences qui sont incluses dans T . On a vu que $\sigma(s) \in \hat{T}$ équivaut à $s \in T$.
- ↪ On a prouvé que deux états d'une même classe d'équivalence étaient envoyés dans une même classe d'équivalence. On peut donc définir $\hat{\delta}$ de $\hat{S} \times \mathcal{A}$ vers \hat{S} par $\hat{\delta}(\sigma(s), x) = \sigma(s.x)$

Ex. 8 Automate réduit

Montrer que $\hat{Q} = (\mathcal{A}, \hat{S}, \hat{\delta}, \sigma_0, \hat{T})$ est un automate déterministe complet émondé qui reconnaît L et qui est de taille minimale.

On voit donc qu'il suffit de déterminer les états équivalents pour réduire l'automate à sa forme minimale. Mais le problème reste le même : on a besoin d'un test d'égalité de langages.

Pour déterminer les classes d'équivalence on va les calculer pas-à-pas en calculant des relations d'équivalences de moins en moins grossières.

Pour tout langage L on définit $L^{(n)} = \{u \in L ; |u| \leq n\}$, c'est l'ensemble des mots de L de longueur inférieure ou égale à n .

Équivalences restreintes

On dit que deux états s et t sont équivalents à l'ordre n si $(L_s)^{(n)} = (L_t)^{(n)}$: on note $s \equiv_n t$ cette relation.

Deux états sont équivalents à l'ordre n si les mots u de longueur n au plus qui envoient les sommets dans T sont les mêmes.

Ex. 9 Calculs des relations

1. Prouver que les classes d'équivalence pour \equiv_0 sont F et $S \setminus F$.
2. Prouver que $[s \equiv_{k+1} t] \iff [s \equiv_k t \text{ et } \forall x \in \mathcal{A} s.x \equiv_k t.x]$.

Ex. 10 Nombre de relations d'équivalence

1. Montrer que si \equiv_{k+1} et \equiv_k sont identiques alors elles sont la relation \equiv .
2. Montrer que, si n est la taille de l'automate, \equiv_{n-2} est la relation \equiv .

Chapitre **X**

Réponses aux exercices

1	Arbres binaires	169
2	Tas	176
3	Logique	185
4	Graphes	192
5	Graphes Valués	205
6	Compléments : Flots	210
7	Langages rationnels	218
8	Automates	227
9	Compléments : automates	235

Dans ce chapitre sont proposées des réponses aux exercices.

Ces solutions n'ont pas la prétention d'être optimales, le but est de montrer que l'on peut arriver à écrire une solution en utilisant les éléments du cours. Parfois plusieurs réponses sont proposées, cela correspond à des méthodes différentes mais parfois à des lectures distinctes de l'énoncé.

Il est recommandé de chercher les exercices avant d'en lire la solution.

1 Arbres binaires

Éléments de réponse pour l'exercice 1

Sens inverse

On suppose que le parcours infixe lit les clés dans l'ordre croissant

Pour tout sous-arbre les clés du fils gauche sont lues par le parcours infixe avant la clé de la racine donc, comme le parcours est croissant, les clés du fils gauche sont inférieures à celle de la racine.

De même les clés du fils droit sont supérieures à celle de la racine.

L'arbre est donc bien un A.B.R.

Sens direct

On vérifie la réciproque par récurrence sur la hauteur.

→ C'est évident pour un sous-arbre de hauteur 0.

→ Si c'est vrai pour tous les arbres de hauteur p au plus. Les fils droit et gauche d'un arbre de hauteur $p+1$ ont une hauteur inférieure ou égale à p . Le parcours infixe de T se fait en parcourant le fils gauche, qui donne des clés croissantes par hypothèse de récurrence, puis en lisant la clé de la racine, supérieure aux précédentes, puis en parcourant le fils droit, toujours en croissant.

Le parcours infixe est croissant.

Éléments de réponse pour l'exercice 2

Le principe est que si on suit un arbre binaire de recherche u_0, u_1, \dots, u_p $u_{i+1} > u_i$ impose $u_j > u_i$ pour tout $j > i$.

1. 122 n'est pas possible après 222, 457
2. 545, 215, 333, 401, 422, 417
3. 601, 403, 555, 408, 523, 411, 466, 417
4. 601, 403, 555, 408, 563, 411, 466, 417
5. 901, 403, 555, 411, 466, 417

Éléments de réponse pour l'exercice 3

Noeud(g , r , d) est un arbre binaire de recherche si

g et d sont des arbres binaires de recherche,

le maximum des clés de g est strictement inférieur à n ,

le minimum des clés de d est strictement supérieur à n .

On peut calculer les maximums et minimums par des fonctions auxiliaire mais cela revient à parcourir tout l'arbre plusieurs fois. On va tout calculer d'un coup.

L'arbre vide n'a pas de maximum, on le remplace par l'entier minimum : tout entier lui sera supérieur. De même le minimum de l'arbre vide sera choisi comme l'entier maximum.

```

let estABR arbre =
  let rec aux a =
    match a with
    |Vide -> (max_int, true, min_int)
    |Noeud(g, r, d) -> let (m1, b1, M1) = aux g in
                       let (m2, b2, M2) = aux d in
                       (m1, b1 && b2 && (M1 < cle r) && (cle r < m2),
M2)
  in let (_,b,_) = aux arbre in b;;

```

Éléments de réponse pour l'exercice 4

```

let rec tranche arbre a b =
  match arbre with
  |Vide -> Vide
  |Noeud(g, r, d) when cle r < a -> tranche d a b
  |Noeud(g, r, d) when cle r > b -> tranche g a b
  |Noeud(g, r, d) -> Noeud(tranche g a b, r, tranche d a b);;

```

Éléments de réponse pour l'exercice 5

L'arbre sert de réservoir qui trie tout seul les éléments qu'on lui ajoute.

→ On insère (dans le désordre) les éléments à trier dans un arbre à partir de l'arbre vide.

→ On lit l'arbre dans un parcours infixe : la liste est triée.

La complexité des insertions est de l'ordre de $\sum_{i=1}^{n-1} \ln(i) \leq n \log_2(n)$.

Le parcours est de complexité linéaire donc le tri est un $\mathcal{O}(n \log_2(n))$.

Éléments de réponse pour l'exercice 6

Si le premier arbre est non vide on coupe le second selon la racine du premier et on ajoute les morceaux aux deux fils.

```

let rec fusion arbre1 arbre2 =
  match arbre1 with
  |Vide -> arbre2
  |Noeud(g, r, d) -> let (pt, gd) = decoupage (cle r) arbre2 in
                     Noeud(fusion g pt, r, fusion d gd);;

```

Éléments de réponse pour l'exercice 7

1. Deux nœuds de même profondeur définissent deux sous-arbres disjoints : ils ne peuvent donc pas être en même temps ancêtres communs de n_1 et n_2 .
2. Les ancêtres d'un d'un ancêtre commun sont des ancêtres communs, en particulier les ancêtres du PPAC sont des ancêtres communs.
Inversement, si n est un ancêtre commun de n_1 et n_2 , sa profondeur h est inférieure à celle du PPAC. On considère alors l'ancêtre de profondeur h du PPAC ; n' . n' est un ancêtre commun de n_1 et n_2 de même hauteur que n donc $n = n'$ d'après la première question. Ainsi n est un ancêtre du PPAC
Les ancêtres communs sont bien les ancêtres du PPAC.
3. On suppose que n_1 n'est pas un descendant de n_2 et n_2 n'est pas un descendant de n_1 .
 n est le PPAC de n_1 et n_2 , on a $n \neq n_1$ et $n \neq n_2$.
Ainsi n_1 et n_2 appartiennent aux fils de n .
S'il appartenait au même fils celui-ci serait un ancêtre commun, ce qui contredirait la minimalité de la hauteur. Ainsi n_1 et n_2 appartiennent à des fils distincts de n .
- 4.

```
let rec ppac arbre p q =
  match arbre with
  | Vide -> failwith "Les elements ne sont pas dans l'arbre"
  | Noeud(g, n, x, d) when n < p && n < q -> ancetreCommun d p q
  | Noeud(g, n, x, d) when n > p && n > q -> ancetreCommun g p q
  | Noeud(g, n, x, d) -> n, x;;
```

Éléments de réponse pour l'exercice 8

1. Comme le parcours infixe visite un nœud juste après son fils gauche on en déduit que le prédécesseur appartient au fils gauche.

Si le prédécesseur avait un fils droit les clé des nœuds de ce fils droit celui-ci serait supérieures tout en restant inférieures à la clé du nœud initial, ce qui contredit la définition du prédécesseur. Ainsi le prédécesseur n'a pas de fils droit.

De même pour le cas symétrique.

2. On peut utiliser l'exercice précédent.

Prouvons plus simplement l'existence du PPAC.

On suit les parcours des sommets depuis la racine :

$$r = a_0 \rightarrow a_1 \rightarrow \dots \rightarrow a_{p-1} \rightarrow a_p = a$$

$$r = b_0 \rightarrow b_1 \rightarrow \dots \rightarrow b_{q-1} \rightarrow b_q = a.$$

Si k est le dernier indice tel que $a_k = b_k$ on a 3 possibilités.

↪ $k = p$, dans ce cas b est un descendant de a , comme b est le successeur il est dans le fils droit et, d'après la question précédente, il est le nœud de clé minimale dans ce fils droit.

↪ Symétriquement, si $k = q$ alors a est le nœud de clé maximale dans le fils gauche de b .

↪ Il reste le cas $k < \min(p, q)$.

Alors a et b appartiennent à deux fils distincts de $c = a_k = b_k$ donc la clé de c est comprise strictement entre celle de a et de b , ce qui est impossible.

3. On commence par le minimum d'un arbre

```
let rec minArbre arbre =
  match arbre with
  |Vide -> failwith "Arbre vide"
  |Noeud(Vide, r, d) -> cle r
  |Noeud(g, r, d) -> minArbre g;;
```

```
let rec suivant k arbre =
  match arbre with
  |Vide -> failwith "Les clé sont trop petites"
  |Noeud(g, r, d) when cle r = k -> minArbre d
  |Noeud(g, r, d) when cle r < k -> suivant k d
  |Noeud(Vide, r, d) -> cle r
  |Noeud(g, r, d) when maxArbre g <= k -> cle r
  |Noeud(g, r, d) -> suivant k g;;
;;
```

Éléments de réponse pour l'exercice 9

1. Réflexivité et transitivité sont immédiates.

La symétrie peut se prouver par récurrence sur la taille ou sur la hauteur.

2. Par récurrence sur la taille ou sur la hauteur ou par induction structurelle.

3.

```
let rec est_prefixe a1 a2 =
  match a1, a2 with
  | Vide, _ -> true
  | Noeud(g1, r1, d1), Noeud(g2, r2, d2) when r1 = r2
    -> (est_prefixe g1 g2) && (est_prefixe d1 d2)
  | _ -> false;;
```

Éléments de réponse pour l'exercice 10

Les ordres d'insertions possibles sont : 6, 4, 2, 3, 8, 7

6, 4, 2, 8, 3, 7

6, 4, 2, 8, 7, 3

6, 4, 8, 2, 3, 7

6, 4, 8, 2, 7, 3

6, 4, 8, 7, 2, 3

6, 8, 4, 2, 3, 7

6, 8, 4, 2, 7, 3

6, 8, 4, 7, 2, 3

6, 8, 7, 4, 2, 3

On prouve la propriété par récurrence sur la taille n .

La propriété est simple pour $n = 1$: il y a une seule manière de construire l'arbre.

On suppose que la propriété est vraie pour tous les arbres (non vides) de taille $k < n$.

$T = \text{Noeud}(g, x, s)$ est un arbre de taille n avec g de taille p et d de taille q .

Les sous-arbres de T sont les sous-arbres de d , les sous-arbres de g et T .

On suppose d'abord que g et d sont non vides.

Pour engendrer T on doit commencer par x puis on doit engendrer d et g en choisissant un remplissage de chacun et l'ordre d'insertion des éléments à droite et à gauche.

1. Il y a $\frac{p!}{\prod_{t \in g} |t|!}$ façon d'engendrer g .

2. Il y a $\frac{q!}{\prod_{t \in d} |t|!}$ façon d'engendrer d .

3. Pour distribuer les insertions à droite et à gauche on doit choisir les p occurrences d'ajout à gauche parmi les $p + q$: il y a donc $\frac{(p + q)!}{p!q!} = \frac{(n - 1)!}{p!q!}$ telles distributions.

$$\text{On trouve donc } \frac{p!}{\prod_{t \in g} |t|!} \frac{q!}{\prod_{t \in d} |t|!} \frac{(n - 1)!}{p!q!} = \frac{(n - 1)!}{\prod_{t \in g} |t|! \prod_{t \in d} |t|!} = \frac{n!}{|T| \prod_{t \in g} |t|! \prod_{t \in d} |t|!} = \frac{n!}{\prod_{t \in T} |t|!}.$$

La propriété est donc vraie pour T .

Si d (ou g) est vide on construit T en insérant x puis les éléments de g (ou d) : il y a le même nombre de construction et $\frac{(n - 1)!}{\prod_{t \in g} |t|!} = \frac{n!}{|T| \prod_{t \in g} |t|!} = \frac{n!}{\prod_{t \in T} |t|!}$

La propriété est encore vraie pour T , c'est-à-dire pour tous les arbres de taille n .

On a prouvé la récurrence.

Éléments de réponse pour l'exercice 11

1. Il y a $(n - 1)!$ permutations telles que l'image du premier terme soit i car on doit ensuite choisir une permutation des $n - 1$ termes restants. La probabilité que le premier terme soit i est donc $\frac{(n-1)!}{n!} = \frac{1}{n}$: la probabilité est uniforme.

2. Quand on recherche un élément de clé k dans un arbre de racine i , 3 cas sont possible selon la valeur de k .

1. Si $k = i$ on ne fait qu'une comparaison. La probabilité de $k = i$ est $\frac{1}{n}$.

2. Si on a $k < i$ (pour $i \geq 2$) on fait deux comparaisons puis on cherche l'élément dans un arbre de taille $i - 1$. La probabilité de cet événement est $\frac{i-1}{n}$.

3. Si on a $k > i$ (pour $i \neq n - 1$) on fait deux comparaisons puis on cherche l'élément dans un arbre de taille $n - i$. La probabilité de cet événement est $\frac{n-i}{n}$.

L'espérance est donc $\frac{1}{n} + \frac{i-1}{n}(2 + a_{i-1}) + \frac{n-i}{n}(2 + a_{n-i})$.

Pour $i = 1$ ou $i = n$ on trouver la valeur $\frac{1}{n} + \frac{n-1}{n}(2 + a_{n-1})$.

3. On a $a_i = \sum_{i=1}^n \frac{1}{n} a_{i,n}$ d'où, en regroupant,

$$\begin{aligned} a_i &= \sum_{i=1}^n \frac{1}{n^2} + \sum_{i=2}^n \frac{i-1}{n^2} (2 + a_{i-1}) + \sum_{i=1}^{n-1} \frac{n-i}{n^2} (2 + a_{n-i}) \\ &= n \frac{1}{n^2} + \frac{2}{n^2} \sum_{i=2}^n (i-1) + \frac{2}{n^2} \sum_{i=1}^{n-1} (n-i) + \frac{1}{n^2} \sum_{i=2}^n (i-1) a_{i-1} + \frac{1}{n^2} \sum_{i=1}^{n-1} (n-i) a_{n-i} \\ &= \frac{1}{n} + \frac{2}{n^2} \sum_{p=1}^{n-1} p + \frac{2}{n^2} \sum_{p=1}^{n-1} p + \frac{1}{n^2} \sum_{p=1}^{n-1} p a_p + \frac{1}{n^2} \sum_{p=1}^{n-1} p a_p \\ &= \frac{1}{n} + \frac{4}{n^2} \frac{n(n-1)}{2} + 2 \frac{1}{n^2} \sum_{p=1}^{n-1} p a_p = 2 - \frac{1}{n} + \frac{2}{n^2} \sum_{p=1}^{n-1} p a_p \end{aligned}$$

4. En utilisant l'indication proposée on obtient

$$n^2 a_n - (n-1)^2 a_{n-1} = 2n^2 - n + 2 \sum_{p=1}^{n-1} p a_p - \left(2(n-1)^2 - (n-1) + 2 \sum_{p=1}^{n-2} p a_p \right)$$

$$= 4n - 3 + 2(n-1)a_{n-1} \text{ pour } n \geq 2$$

On obtient donc $a_n = \frac{1}{n^2} (4n - 3 + 2(n-1)a_{n-1} + (n-1)^2 a_{n-1})$

d'où $a_n = \frac{1}{n^2} (4n - 3 + (n^2 - 1)a_{n-1})$.

5. On commence par étudier les suites définies par la formule homogène

$$u_n = \frac{1}{n^2} ((n^2 - 1)u_{n-1}) = \frac{(n+1)(n-1)}{n^2} u_{n-1}$$

$$\text{Ainsi } u_n = \frac{(n+1) \cdots 3 \cdot (n-1) \cdots 1}{(n(n-1) \cdots 2)^2} u_1 = \frac{n+1}{2n} u_1 = K \frac{n+1}{n}.$$

On pose alors (en faisant varier la constante) $a_n = \frac{n+1}{n} b_n$.

La récurrence devient $\frac{n+1}{n} b_n = \frac{1}{n^2} (4n - 3 + (n^2 - 1) \frac{n}{n-1} b_{n-1})$ donc

$$b_n = \frac{4n - 3}{n(n+1)} + b_{n-1} = \frac{7}{n+1} - \frac{3}{n} + b_{n-1}.$$

$$\text{On a alors } b_n = \sum_{k=2}^n \frac{7}{k+1} - \frac{3}{k} + b_1 = \frac{7}{n+1} + 4 \sum_{k=1}^n \frac{1}{k} - \frac{15}{2} + b_1.$$

$$\text{Comme on a } b_1 = \frac{1}{2} a_1 = \frac{1}{2} \text{ on a } a_n = \frac{7}{n} + \frac{4(n+1)}{n} \sum_{k=1}^n \frac{1}{k} - 7 \frac{n+1}{n}.$$

$$\text{Ainsi } a_n = \frac{4(n+1)}{n} \sum_{k=1}^n \frac{1}{k} - 7 = 4 \ln(n).$$

2 Tas

Éléments de réponse pour l'exercice 1

Un tableau est un tas si tous les fils ont une clé supérieure à celle de leur père. Plutôt que tester les fils d'un nœud, il faudrait s'assurer de leur existence, on va tester le père de chaque nœud.

```
let estTas tas =
  let n = tas.taille in
  let a = tas.arbre in
  let valide = ref true in
  for i = 1 to (n-1) do
    if a.(i) >= a.((i-1)/2) then valide := false done;
  !valide;;
```

Éléments de réponse pour l'exercice 2

Pour enlever l'élément d'indice k il suffit de reproduire l'extraction du minimum :

```
let enlever k tas =
  let n = tas.taille - 1 in
  echange tas k n;
  tas.taille <- n;
  descendre k tas;;
```

Éléments de réponse pour l'exercice 3

```
let priorite x = x;;

let mettreEnTas t =
  let n = Array.length t in
  let tas = {taille = n; arbre = t} in
  for i = 0 to (n-1) do remonter i tas done;
  tas;;
```

La complexité est majorée par $\sum_{i=1}^{n-1} \lfloor \log_2(i+1) \rfloor$: c'est un $\mathcal{O}(n \log(n))$.

Éléments de réponse pour l'exercice 4

```
let triParTas t =
  let tas = mettreEnTas t in
  let n = tas.taille in
  for i = 0 to (n-1) do enlever tas done;;
```

Le tableau t est alors trié.

```
let t = [|5; 12; 9; 1; 8; 13; 4|];;
triParTas t;;
t;;

# - : int array = [|1; 4; 5; 8; 9; 12; 13|]
```

La complexité est toujours un $\mathcal{O}(n \log(n))$.

Éléments de réponse pour l'exercice 5

Le dernier fils admet $n-1$ pour indice donc le dernier père admet $(n-2)/2$ pour indice.

```
let mettreEnTas t =
  let n = Array.length t in
  let m = (n-2)/2 in
  let tas = {taille = n; arbre = t} in
  for i = 0 to m do descendre (m-i) tas done;
  tas;;
```

Il y a 2^k nœuds de profondeur k pour $k < h$, hauteur de l'arbre.

Pour chacun de ces nœuds la descente demande au plus $h - k$ étapes.

La complexité est donc majorée par $\sum_{k=0}^{h-1} (h-k)2^k$.

Si on pose $S_p = \sum_{k=0}^{p-1} (p-k)2^k$ on a

$$S_{p+1} = \sum_{k=0}^p (p+1-k)2^k = 2^p + \sum_{k=0}^{p-1} ((p-k)2^k + 2^k) = 2^p + S_p + \sum_{k=0}^{p-1} 2^k = 2.2^p + S_p - 1.$$

Comme on a $S_0 = 0$, on en déduit $S_p = \sum_{k=0}^{p-1} 2.2^k - 1 = 2.(2^p - 1) - n$.

Ainsi S_h est majoré par $2.2^h - h - 2$ avec $2^h \leq n < 2^{h+1}$ donc $S_h \leq 2n$: la complexité est un $\mathcal{O}(n)$.

Éléments de réponse pour l'exercice 6

1. `estVide` est inchangés.

2.

```
let fpVide a = {taille = 0;
                arbre = Array.make nMax a;
                position = Array.make nMax (-1)};;
```

3. La fonction `present` est assez simple.

```
let present index t = (t.(index) <> -1);;
```

4. La fonction `echange` doit maintenant travailler sur les deux tableaux

```
let echange tas i j =
  let t = tas.arbre and p = tas.position in
  let x = t.(i) and y = t.(j) in
  t.(i) <- y;
  t.(j) <- x;
  p.(indice x) <- j;
  p.(indice y) <- i;;
```

5. `premier`, `remonter`, `filGrand` et `descendre` sont inchangés.

6. On n'ajoute pas un élément si son indice est déjà présent.

```
let ajouter x tas =
  if not (present (indice x) t)
  then begin
    let n = tas.taille in
    tas.arbre.(n) <- x;
    tas.taille <- n + 1;
    tas.position.(indice x) <- n;
    monter n tas end;;
```

7. Quand on enlève l'élément prioritaire, on supprime aussi sa position.

```

let enlever tas =
  let n = tas.taille - 1 in
  echange tas 0 n;
  tas.taille <- n;
  t s.position.(indice t.arbre.(n)) <- -1;
  descendre 0 tas;;

```

8. On peut alors modifier la clé. Il faudra alors soit remonter soit descendre l'élément.

```

let changerPrio index valeur tas =
  if present index tas
  then let k = tas.position.(index) in
        tas.arbre.(k) <- newPrio (tas.arbre.(k)) valeur;
        descendre k tas;
        remonter k tas;;

```

Éléments de réponse pour l'exercice 7

1. Pour transformer un élément il suffit de le chercher et de transformer le nœud :

```

let rec change k x arbre =
  match arbre with
  |Vide -> if k = 1
            then Noeud(Vide, x, Vide)
            else failwith "Position non accessible"
  |Noeud(g,n,d) -> if k = 1
                    then Noeud(g,x,d)
                    else if k mod 2 = 0
                          then Noeud(change (k/2) x g ,n, d)
                          else Noeud(g, n, change (k/2) x d);;

```

Comme suggéré par l'énoncé on autorise, si on parvient à un arbre vide, l'ajout d'un élément d'indice k si l'élément d'indice $k/2$ est défini. En effet on ne peut appeler `change 1 x Vide` que depuis un arbre non vide⁵.

2. On peut créer un tableau à partir du vide en ajoutant un-à-un tous les éléments grâce à la possibilité de la question précédente. Si on effectue cela récursivement on conserve en fait tous les

⁵ ou directement en voulant changer une valeur d'un tableau vide

tableaux intermédiaires. On va donc utiliser une boucle et modifier une variable référencée : cela permet au **garbage collector** de libérer la place.

```
let creeTableau n x =
  let a = ref Vide in
  for i = 1 to n do a := change i x !a done;
  !a;;
```

On verra que les tableaux fonctionnels ont une structure équilibrée et que les fils sont similaires, le fils gauche ayant un terme de plus. On peut retrouver un algorithme récursif

```
let rec creeTableau n x =
  if n = 0
  then Vide
  else let k = (n-1)/2 in
        Noeud(creeTableau (n-1-k) x,x,creeTableau k x));;
```

3. Comme indiqué ci-dessus, un tableau de taille $n \leq 2^h - 1$ est représenté par un arbre de hauteur h donc les parcours de lecture ou de changement ont une complexité en $O(h) = O(\log(n))$.

Éléments de réponse pour l'exercice 8

1. L'arbre de gauche est défini par les positions k pour k pair. De plus, depuis la racine du fils gauche, on définit les positions à l'aide de $k/2$; on remplit donc un tableau fonctionnel avec les positions $2/2, 4/2, \dots, n/2$. On construit ainsi un arbre de Braun.
De même pour le fils droit.
2. D'après la question précédente il y a $n/2 = \lfloor \frac{n}{2} \rfloor$ nœuds dans le fils gauche.
La racine n'est dans aucun fils donc il y a $n - 1 - \lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil - 1$ nœuds dans le fils droit.
On a $\lfloor \frac{n}{2} \rfloor \leq \lceil \frac{n}{2} \rceil \leq \lfloor \frac{n}{2} \rfloor + 1$ d'où le résultat.
3. Il n'y a qu'un arbre de Braun de taille p .
De plus l'arbre de taille $p + 1$ est obtenu en ajoutant un élément.
Le fils gauche est ainsi égal au fils droit ou est obtenu en ajoutant un nœud ce qui ne peut augmenter la hauteur que de 1 au maximum.
4. La hauteur est atteinte pour le fils gauche d'après la troisième question.
Par récurrence on en déduit que la feuille la plus à gauche est à la hauteur maximale.

Éléments de réponse pour l'exercice 9

1. On va montrer, par récurrence sur n que la fonction `ajouterBraun x t` appliquée à un arbre qui représente un tableau fonctionnel d'indices de 1 à n le transforme en un tableau fonctionnel d'indices de 1 à $n + 1$.

→ C'est évident pour $n = 0$.

→ On suppose que cela est vrai pour tous les entiers $k \leq n - 1$ avec $n \geq 2$.

t est un arbre de Braun de taille n , $t = \text{Noeud}(g, y, d)$, avec g et d de taille respectives t_g et t_d .

On note $t' = \text{ajouterBraun } x \ d$; c'est, d'après l'hypothèse de récurrence, un arbre de Braun.

Si d et g ont la même taille p alors t' et g sont des arbres de Braun de tailles respectives $p + 1$ et p donc $\text{Noeud}(t', y, g)$ a la structure d'un arbre de Braun de taille $2p + 1 = n + 1$.

De même si g a un élément de plus que d .

2. La réciproque de "ajouter à droite puis inverser les fils" est "inverser les fils puis retrancher à droite" c'est-à-dire "retrancher à gauche puis inverser les fils".

La fonction renvoie la valeur enlevée en plus du nouvel arbre.

```
let rec enleverBraun t =
  match t with
  | Vide -> failwith "Il n'y a rien à enlever"
  | Noeud(Vide,x,_) -> (x,Vide) (* Si g est vide alors d l'est aussi*)
  | Noeud(g,x,d) -> let (y,g') = enleverBraun g in
                    (y,Noeud(d,x,g'));;
```

Éléments de réponse pour l'exercice 10

1. On envoie vers le bas la plus petite clé :

```
let rec ajouterTasBraun x t =
  match t with
  | Vide -> Noeud(Vide,x,Vide)
  | Noeud(g,y,d) -> if cle x < cle y
                    then Noeud(ajouterTasBraun x d,y,g)
                    else Noeud(ajouterTasBraun y d,x,g);;
```

2. Pour enlever le plus petit élément, celui qui est à la racine, on peut

→ enlever le dernier élément, en conservant sa valeur,

→ le placer à la racine

→ le descendre en échangeant avec le plus grand de ses fils, récursivement.

On commence par écrire une fonction qui fait descendre un nœud de clé trop petite vers le bas de l'arbre. On supposera que l'arbre vérifie toutes les propriétés à l'exception de la clé de la racine qui peut être trop petite.

```

let rec descendre t =
  match t with
  |Vide -> t
  |Noeud(Vide,_,_) -> t
  |Noeud(Noeud(g,y,d),x,Vide)
  -> if x < y then t
      else Noeud(descendre(Noeud(g,x,d)),y,Vide)
  |Noeud(Noeud(g,y,d),x,Noeud(g',y',d'))
  -> if y < y'
      then (if x > y
            then Noeud(descendre(Noeud(g,x,d)),y,Noeud(g',y',d'))
            else t)
      else (if x > y'
            then Noeud(Noeud(g,y,d),y',descendre(Noeud(g',x,d')))
            else t);;

```

Il suffit alors de remplacer la racine par le dernier terme et de faire descendre.

```

let resteTasBraun t =
  let (x,t') = enleverBraun t in
  match t' with
  |Vide -> Vide
  |Noeud(g,y,d) -> descendre(Noeud(g,x,d));;

```

Éléments de réponse pour l'exercice 11

1. Dans un arbre de rang r toutes les feuilles sont à une profondeur au moins $r - 1$ donc tous les niveaux de 0 à $r - 1$ sont remplis.
Il y a donc au moins $1 + 2 + \dots + 2^{r-1} = 2^r - 1$ nœuds.
2. Le rang d'un arbre est le maximum des rangs de ses fils augmenté de 1 donc le rang d'un arbre gauchiste est le rang de son fils droit augmenté de 1. Le résultat demandé se prouve alors par récurrence sur le rang.
On en déduit que $r = \mathcal{O}(\log_2(n))$ où n est la taille de l'arbre.
3. Une fusion peut se faire de manière récursive en choisissant la plus petite racine des deux arbres et en fusionnant l'autre arbre avec un fils du premier (dont la racine a été choisie). Ici on choisit

systématiquement le fils droit. On n'est pas certain que le résultat vérifie la propriété de plus petit rang à droite donc on crée l'arbre en fonction de la comparaison des rangs.

```

let rang arbre =
  match arbre with
  |Vide -> 0
  |Noeud(r,_,_,_) -> r;;

let faireArbreG racine fils1 fils2 =
  let r1 = rang fils1 in
  let r2 = rang fils2 in
  if r1 < r2
  then Noeud(r1+1, fils2, racine, fils1)
  else Noeud(r2+1, fils1, x, fils2);;

let rec fusion a1 a2 =
  match a1, a2 with
  |Vide, _ -> a2
  |_, Vide -> a1
  |Noeud(_,g1,x1,d1), Noeud(_,g2,x2,d2) when x1 > x2
    -> faireArbreG x1 g1 (fusion d1 a2)
  |Noeud(_,g1,x1,d1), Noeud(_,g2,x2,d2)
    -> faireArbreG x2 g2 (fusion d2 a1);;

```

La complexité est proportionnelle au nombre d'appels récursifs donc est un $\mathcal{O}(r_1 + r_2)$.

4.

```
let fpVide () -> Vide;;

let estVide f = (f = Vide);;

let ajouter x f =
  fusion f (Noeud(1, Vide, x, Vide));;

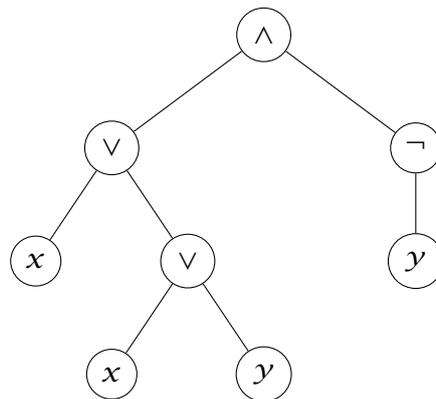
let premier f =
  match f with
  | [] -> failwith ''La file est vide''
  | Noeud(_,_,x,_) -> x;;

let enlever f =
  match f with
  | [] -> failwith ''La vile est vide''
  | Noeud(_,g,_,d) -> fusion g d;;
```

Les opérations sont bien en $\mathcal{O}(\log_2(n))$ d'après le résultat du 2.

3 Logique

Éléments de réponse pour l'exercice 1



Éléments de réponse pour l'exercice 2

$((x \wedge (z \vee t)) \vee (y \vee (\neg z)))$

Éléments de réponse pour l'exercice 3

Le résultat est **F**.

Éléments de réponse pour l'exercice 4

- ↪ On a $\neg(\neg V) = \neg F = V$ et $\neg(\neg F) = \neg V = F$; pour toute valuation v ,
- ↪ G est une tautologie donc $\tilde{v}(G) = V$ pour toute valuation v .
On a $F \wedge V = F$ et $V \wedge V = V$; ainsi, pour toute valuation v ,
 $\tilde{v}(F \wedge G) = \tilde{v}(F) \wedge \tilde{v}(G) = \tilde{v}(F) \wedge V = \tilde{v}(F)$ d'où l'équivalence $F \wedge G \equiv F$.
- ↪ G est une contradiction donc $\tilde{v}(G) = F$ pour toute valuation v .
On a $F \vee F = F$ et $V \vee F = V$; ainsi, pour toute valuation v ,
 $\tilde{v}(F \vee G) = \tilde{v}(F) \vee \tilde{v}(G) = \tilde{v}(F) \vee F = \tilde{v}(F)$ d'où l'équivalence $F \vee G \equiv F$.

Éléments de réponse pour l'exercice 5

F	G	$G \wedge H$	$\neg(F \wedge G)$	$\neg F$	$\neg G$	$\neg F \vee \neg G$
V	V	V	F	F	F	F
V	F	F	V	F	V	V
F	V	F	V	V	F	V
F	F	F	V	V	V	V

On a bien l'équivalence $\neg(F \wedge G) \equiv \neg F \vee \neg G$.

F	G	$G \vee H$	$\neg(F \vee G)$	$\neg F$	$\neg G$	$\neg F \wedge \neg G$
V	V	V	F	F	F	F
V	F	V	F	F	V	F
F	V	V	F	V	F	F
F	F	F	V	V	V	V

On a bien l'équivalence $\neg(F \vee G) \equiv \neg F \wedge \neg G$.

Éléments de réponse pour l'exercice 6

On calcule les tables de vérités

F	G	H	$G \vee H$	$F \wedge (G \vee H)$	$F \wedge G$	$F \wedge H$	$F \wedge G \vee F \wedge H$
V	V	V	V	V	V	V	V
V	V	F	V	V	V	F	V
V	F	V	V	V	F	V	V
V	F	F	F	F	F	F	F
F	V	V	V	F	F	F	F
F	V	F	V	F	F	F	F
F	F	V	V	F	F	F	F
F	F	F	F	F	F	F	F

On a bien l'équivalence $F \wedge (G \vee H) \equiv F \wedge G \vee F \wedge H$.

F	G	H	$G \wedge H$	$F \vee G \wedge H$	$F \vee G$	$F \vee H$	$(F \vee G) \wedge (F \vee H)$
V	V	V	V	V	V	V	V
V	V	F	F	V	V	V	V
V	F	V	F	V	V	V	V
V	F	F	F	V	V	V	V
F	V	V	V	V	V	V	V
F	V	F	F	F	V	F	F
F	F	V	F	F	F	V	F
F	F	F	F	F	F	F	F

On a bien l'équivalence $F \vee G \wedge H \equiv (F \vee G) \wedge (F \vee H)$.

Éléments de réponse pour l'exercice 7

L'idempotence découle de $V \wedge V = V$ et $F \wedge F = F$.

F	G	$F \vee G$	$\neg(F \vee G)$	$F \wedge (F \vee G)$
V	V	V	F	F
V	F	V	F	F
F	V	V	F	V
F	F	F	V	V

On a bien l'équivalence $F \wedge (F \vee G) \equiv F$.

Éléments de réponse pour l'exercice 8

Pour toute valuation v et toute variable x_i , on a $\tilde{v}(l_v(x_i)) = \mathbf{V}$ d'où $\tilde{v}(D_v) = \mathbf{V}$.

Pour deux valuations v et v' distinctes il existe i tel que $v'(x_i) = \neg v(x_i)$;

on a alors $\tilde{v}(\neg x_i) = \neg \tilde{v}'(\neg x_i)$ d'où $\tilde{v}'(l_v(x_i)) = \neg \tilde{v}(l_v(x_i)) = \mathbf{F}$.

On a alors $\tilde{v}'(D_v) = \mathbf{F}$.

Si v_0 appartient à V_F l'une des clauses de F_N est D_{v_0} donc $\tilde{v}_0(F_N)$, qui est la disjonction des $\tilde{v}_0(D_v)$, vaut **V**. On a aussi $\tilde{v}_0(F) = \mathbf{V}$ car $v_0 \in V_F$.

Si v_0 n'appartient pas à V_F toutes les clauses de F_N donnent une valeur **F** par \tilde{v}_0 donc $\tilde{v}_0(D_v)$ vaut **F**. On a aussi $\tilde{v}_0(F) = \mathbf{F}$ car $v_0 \notin V_F$.

Ainsi F et F_N ont la même valeur pour toute valuation, elles sont donc équivalentes.

Éléments de réponse pour l'exercice 9

On note φ la fonction associée à la table.

$\varphi(\mathbf{V}, \mathbf{V})$	$\varphi(\mathbf{V}, \mathbf{F})$	$\varphi(\mathbf{F}, \mathbf{V})$	$\varphi(\mathbf{F}, \mathbf{F})$	$\varphi(x, y)$
V	V	V	V	$x \vee \neg x$
V	V	V	F	$x \vee y$
V	V	F	V	$x \vee \neg y$
V	V	F	F	x
V	F	V	V	$\neg x \vee y$
V	F	V	F	y
V	F	F	V	$x \Leftrightarrow y$
V	F	F	F	$x \wedge y$
F	V	V	V	$\neg(x \wedge y)$
F	V	V	F	$x \Leftrightarrow \neg y$
F	V	F	V	$\neg y$
F	V	F	F	$x \wedge \neg y$
F	F	V	V	$\neg x$
F	F	V	F	$\neg x \wedge y$
F	F	F	V	$\neg(x \vee y)$
F	F	F	F	$x \wedge \neg x$

Éléments de réponse pour l'exercice 10

1. Si $\tilde{v}(F) = \mathbf{V}$ alors $\tilde{v}(\text{ite}(F, G, H)) = \tilde{v}(G)$, si $\tilde{v}(F) = \mathbf{F}$ alors $\tilde{v}(\text{ite}(F, G, H)) = \tilde{v}(H)$.
2. $F \wedge G \equiv \text{ite}(F, G, F)$, $F \vee G \equiv \text{ite}(F, F, G)$
3. $\neg F \equiv \text{ite}(F, \mathbf{0}, \mathbf{1})$

Éléments de réponse pour l'exercice 11

1. $[F \Rightarrow G] \equiv [\neg F \vee G] \equiv [\neg\neg G \vee \neg F] \equiv [\neg G \Rightarrow \neg F]$
 $[F \Rightarrow (G \Rightarrow H)] \equiv [\neg F \vee (G \Rightarrow H)] \equiv [\neg F \vee (\neg G \vee H)] \equiv [\neg(F \wedge G) \vee H] \equiv [F \wedge G \Rightarrow H]$

2.

F	G	$F \Rightarrow G$	$(F \Rightarrow G) \Rightarrow F$	$((F \Rightarrow G) \Rightarrow F) \Rightarrow F$
\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{V}
\mathbf{V}	\mathbf{F}	\mathbf{F}	\mathbf{V}	\mathbf{V}
\mathbf{F}	\mathbf{V}	\mathbf{V}	\mathbf{F}	\mathbf{V}
\mathbf{F}	\mathbf{F}	\mathbf{V}	\mathbf{V}	\mathbf{V}

3.

p	q	r	$p \Rightarrow q$	$q \Rightarrow r$	F	$p \wedge q$	$(p \vee r)$	G
\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{V}
\mathbf{V}	\mathbf{V}	\mathbf{F}	\mathbf{V}	\mathbf{F}	\mathbf{F}	\mathbf{V}	\mathbf{F}	\mathbf{F}
\mathbf{V}	\mathbf{F}	\mathbf{V}	\mathbf{F}	\mathbf{V}	\mathbf{F}	\mathbf{V}	\mathbf{F}	\mathbf{F}
\mathbf{V}	\mathbf{F}	\mathbf{F}	\mathbf{F}	\mathbf{V}	\mathbf{F}	\mathbf{V}	\mathbf{F}	\mathbf{F}
\mathbf{F}	\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{V}
\mathbf{F}	\mathbf{V}	\mathbf{F}	\mathbf{V}	\mathbf{F}	\mathbf{F}	\mathbf{V}	\mathbf{F}	\mathbf{F}
\mathbf{F}	\mathbf{F}	\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{F}	\mathbf{F}	\mathbf{V}
\mathbf{F}	\mathbf{F}	\mathbf{F}	\mathbf{V}	\mathbf{V}	\mathbf{V}	\mathbf{F}	\mathbf{F}	\mathbf{V}

Les deux formules sont équivalentes.

Éléments de réponse pour l'exercice 12

Les loi de Morgan donnent

$$[(\neg F \vee G) \wedge (\neg G \vee F)] \equiv [\neg F \wedge \neg G \vee \neg F \wedge F \vee G \wedge \neg G \vee G \wedge F].$$

Or $\neg F \wedge F \equiv \mathbf{F}$ est absorbé dans une conjonction, de même pour $G \wedge \neg G$ d'où

$$[(\neg F \vee G) \wedge (\neg G \vee F)] \equiv [\neg F \wedge \neg G \vee G \wedge F].$$

Si $F \equiv G$ alors, pour toute valuation v , $\tilde{v}(F) = \tilde{v}(G)$ donc

soit $\tilde{v}(F) = \tilde{v}(G) = \mathbf{V}$ d'où $\tilde{v}(F \wedge G) = \mathbf{V}$

soit $\tilde{v}(F) = \tilde{v}(G) = \mathbf{F}$ d'où $\tilde{v}(\neg F \wedge \neg G) = \mathbf{V}$.

Dans les deux cas on a $\tilde{v}(F \wedge G \vee \neg F \wedge \neg G) = \mathbf{V}$.

C'est vrai pour toute valuation donc $F \wedge G \vee \neg F \wedge \neg G$ (c'est-à-dire $F \Leftrightarrow G$) est une tautologie.

Inversement si $F \Leftrightarrow G$ est une tautologie alors elle vraie pour toute valuation ce qui signifie que, pour toute valuation v , $\tilde{v}(F)$ et $\tilde{v}(G)$ sont tous deux vrais ou tous deux faux, ils sont toujours égaux : F et G sont équivalents.

Éléments de réponse pour l'exercice 13

1. $[F|F] \equiv [\neg F \vee \neg F] \equiv [\neg F]$

2. $[F \vee G] \equiv [\neg F | \neg F] \equiv [(F|F)|(G|G)]$.

3. Pour pouvoir transformer toute formule logique il faut encore écrire $F \wedge G$ en fonction uniquement du connecteur de Schaeffer.

$$[F \wedge G] \equiv [\neg(\neg F \vee \neg G)] \equiv [\neg(F|G)] \equiv [(F|G)|(F|G)].$$

4.

```
let rec conversion exp =
  match exp with
  |Variable k -> VarS k
  |Non f -> let ff = conversion f in Shf(ff, ff)
  |Ou(f, g) -> let ff = conversion f in
                let gg = conversion g in
                Shf(Shf(ff, ff), Shf(gg, gg))
  |Et(f, g) -> let ff = conversion f in
                let gg = conversion g in
                Shf(Shf(ff, gg), Shf(gg, gg));;
```

Écrire une fonction qui transforme une formule avec le type défini dans le cours en une formule du nouveau type.

Éléments de réponse pour l'exercice 14

1. A dit $G \vee D$, B dit $\neg D$.

2. C est la propriété $(G \vee D) \wedge \neg D \vee \neg(G \vee D) \wedge \neg \neg D$

3. C doit être vrai, sa table de vérité est

G	D	$G \vee D$	$\neg D$	C
V	V	V	F	F
V	F	V	V	F
F	V	V	F	V
F	F	F	V	F

C ne peut être vrai que si G est vrai et D est faux, il faut prendre le chemin de gauche et les 2 sphinx ont dit la vérité.

Éléments de réponse pour l'exercice 15

On note F_i la propriété "Le processeur i fonctionne normalement".

La première réponse s'interprète sous la forme $F_1 \wedge (F_2 \wedge F_3) \vee \neg F_1 \wedge \neg(F_2 \wedge F_3)$.

Le seconde s'interprète $F_1 \wedge F_2 \vee \neg F_1 \wedge \neg F_2$.

La conjonction des deux réponse doit être vraie

$$F = (F_1 \Leftrightarrow (F_2 \wedge F_3)) \wedge (F_1 \Leftrightarrow \neg F_2)$$

F_1	F_2	F_3	$F_2 \wedge F_3$	$F_1 \Leftrightarrow (F_2 \wedge F_3)$	$F_1 \Leftrightarrow \neg F_2$	F
V	V	V	V	V	F	F
V	V	F	F	F	F	F
V	F	V	F	F	V	F
V	F	F	F	F	V	F
F	V	V	V	F	V	F
F	V	F	F	V	V	V
F	F	V	F	V	F	F
F	F	F	F	V	F	F

F n'est vraie que si F_2 fonctionne et F_1 et F_2 ne fonctionnent pas.

Éléments de réponse pour l'exercice 16

On note K_d la propriété "Un kjalt a un dard",

K_g la propriété "Un kjalt a des griffes"

et K_p la propriété "Un kjalt a des pinces".

On traduit $A = K_d \vee K_g$, $B = \neg K_d$ et $C = K_g \wedge K_p$.

La condition de politesse indique $A \Rightarrow B$ et $A \Rightarrow C$ sont vraies.

K_d	K_g	K_p	A	B	C	$A \Rightarrow B$	$B \Rightarrow C$
V	V	V	V	F	V	F	F
V	V	F	V	F	F	F	V
V	F	V	V	F	F	F	V
V	F	F	V	F	F	F	V
F	V	V	V	V	V	V	V
F	V	F	V	V	F	V	F
F	F	V	F	V	F	V	F
F	F	F	F	V	F	V	F

La règle de politesse n'est vérifiée que si le klalt a des griffes et des pinces.

Dans ce cas les 3 orateurs ont dit la vérité ce qui reste vrai dans la suite.

D'après A un lyop peut être mauve, il ne peut pas être jaune.

D'après B il ne peut pas être vert.

C permet cette couleur mauve, seule possible.

4 Graphes

Éléments de réponse pour l'exercice 1

Chaque arc a une origine et une extrémité.

Si les arêtes sont des ensemble à deux éléments, $\sum_{s \in S} d(s) = 2|A|$.

Éléments de réponse pour l'exercice 2

Par l'absurde : si ce n'est pas le cas alors, comme le degré est compris entre 0 et $n - 1$, tous les degrés sont atteints. Il y a donc un sommet de degré 0 donc sans voisin et un sommet de degré $n - 1$ donc voisin de tous. Il y a contradiction.

Éléments de réponse pour l'exercice 3

Si a et b sont deux sommets de S' il existe un chemin de a vers b et un chemin de b vers a dans S . Si s est un sommet du chemin de a vers b on veut montrer qu'il appartient à S' , c'est-à-dire qu'il est équivalent à a (par exemple).

Le chemin de a vers b permet de définir un chemin de a vers s .

Il définit aussi un chemin de s vers b qu'on peut prolonger par le chemin de b vers a donc il existe un chemin de s vers a : a et s sont équivalents.

Éléments de réponse pour l'exercice 4

On note S_0 la composante connexe de G contenant s_0 .

1. Si s appartient à S_0 alors, comme $s_0 \in S_0$, il existe un chemin entre s_0 et s .

Inversement s'il existe un chemin entre s_0 et s alors, comme le graphe n'est pas orienté, on peut retourner le chemin dans G pour obtenir un chemin de s à s_0 : s et s_0 sont équivalents donc $s \in S_0$.

2. Le graphe induit par S_0 est connexe d'après le théorème.

On va montrer que si S' est telle que $s_0 \in S'$ et le graphe induit par S' est connexe alors $S' \subset S_0$.

En effet les point de S' sont accessibles depuis s_0 .

Éléments de réponse pour l'exercice 5

L'idée est de partir d'un graphe sans arête est d'ajouter les arêtes une à une.

Un graphe sans arête de n sommet admet n composantes connexes.

Chaque arête ajoutée

- soit diminue de 1 le nombre de composantes connexes
- soit relie deux sommets qui sont déjà dans une même composante connexe donc crée un cycle.

Pour obtenir un graphe connexe il faut donc au moins $n - 1$ arêtes.

Éléments de réponse pour l'exercice 6

On note G' le graphe obtenu en ôtant l'arête (x, y) : $G' = (S, A \setminus \{(x, y)\})$.

↔ 1) ⇒ 2) Si (x, y) est un isthme et s'il existait un chemin simple de x à y distinct de (x, y) alors ce chemin est de longueur au moins 2 donc ne peut pas comporter l'arête (x, y) car il est

simple : c'est un chemin dans G' . En remplaçant (x, y) par ce chemin on voit que deux sommets équivalents dans g sont encore équivalents dans G' donc G' est connexe, ce qui est exclu.

(x, y) est le seul chemin simple de G reliant x et y .

↔ 2) ⇒ 1) Si (x, y) est le seul chemin de G reliant x et y alors x et y ne sont pas équivalents dans G' donc G' n'est pas connexe.

↔ 2) ⇔ 3) L'existence d'un cycle simple contenant (x, y) est équivalente à l'existence d'un chemin simple entre x et y distinct de (x, y) .

Éléments de réponse pour l'exercice 7

```
let creerGraphe n = {taille = n; aretes = []};;
```

```
let taille g = g.taille;;
```

```
let estArete g s1 s2 =
  let rec app x liste =
    match liste with
    | [] -> false
    | t::q -> t=x || app x q in
  app (s1, s2) g.aretes;;
```

```
let ajouter g s1 s2 =
  if not (estArete g s1 s2)
  then g.aretes <- (s1, s2)::g.aretes;;
```

```
let retirer g s1 s2 =
  let rec oter x liste =
    match liste with
    | [] -> []
    | t::q when t = x -> q
    | t::q -> t::(oter x q) in
  g.aretes <- oter (s1, s2) g.aretes;;
```

```

let voisins g s =
  let rec vois x liste =
    match liste with
    | [] -> []
    |(a, b)::q when a = x -> b::(vois x q)
    |t::q -> vois x q in
  vois s g.ares;

```

```

let aretes g = g.ares;

```

Éléments de réponse pour l'exercice 8

```

let aretes2matrice g =
  let n = g.taille in
  let gM = Array.make_matrix n n false in
  let rec aux liste =
    match liste with
    | [] -> ()
    |(i,j)::q -> gM.(i).(j) <- true; aux q in
  aux g.ares;
  gM;

```

```

let matrice2aretes g =
  let n = Array.length g in
  let ar = ref [] in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      if g.(i).(j)
      then ar := (i, j)::(!ar) done done;
  {taille = n; aretes = !ar};

```

Éléments de réponse pour l'exercice 9

```

let matrice2tableau g =
  let n = Array.length g in
  let gT = Array.make n [] in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      if g.(i).(j)
      then gT.(i) <- j::(gT.(i)) done done;
  gT;;

```

```

let tableau2matrice g =
  let n = Array.length g in
  let gM = Array.make_matrix n n false in
  let rec ecrit s liste =
    match liste with
    | [] -> ()
    | t::q -> gM.(s).(t) <- true; ecrit s q in
  for i = 0 to (n -1) do
    ecrit i g.(i) done;
  gM;;

```

Éléments de réponse pour l'exercice 10

```

let tableau2aretes g =
  let n = Array.length g in
  let ar = ref [] in
  let rec ajoute s liste =
    match liste with
    | [] -> []
    | t::q -> (s, t)::(ajoute s q) in
  for i = 0 to (n -1) do
    ar := (ajoute i g.(i))@(!ar) done;
  {taille = n; aretes = !ar};;

```

```

let aretes2tableau g =
  let n = g.taille in
  let gT = Array.make n [] in
  let rec aux liste =
    match liste with
    | [] -> ()
    |(i,j)::q -> gT.(i) <- j::gT.(i); aux q in
  aux g.arestes;
  gT;;

```

Éléments de réponse pour l'exercice 11

```

let listeCC g =
  let n = taille g in
  let vus= Array.make n false in
  let resultat = ref [] in
  let rec visiter cc s =
    if not vus.(s)
    then (vus.(s) <- true;
          cc := s::(!cc);
          List.iter (visiter cc) (voisins g s)) in
  for i = 0 to (n-1) do
    if not vus.(i)
    then (let cc = ref [] in
          visiter cc i;
          resultat := !cc::!resultat) done;

```

Éléments de réponse pour l'exercice 12

Quand on ajoute des arêtes une-par une à partir d'un graphe sans arêtes

- soit on diminue de 1 le nombre de composantes connexes
- soit on relie deux sommets qui sont déjà dans une même composante connexe donc on crée un cycle.

Comme on ne peut diminuer que $n - 1$ fois le nombre de composantes connexes, n arêtes vont créer (au moins) un cycle.

Pour faire passer le nombre de composantes connexes de n à 1 il faut au moins $n - 1$ arêtes.

Éléments de réponse pour l'exercice 13

Non, il peut être une union disjointe de cycles.

Si on suppose qu'il est connexe, comme la somme des degrés est $2n$, il admet un cycle. Si ce cycle ne contenait pas tous les sommets alors il devrait y avoir au moins un sommet de degré 3 dans le cycle pour pouvoir joindre un sommet hors du cycle.

Éléments de réponse pour l'exercice 14

La propriété : G est un arbre est notée 0.

↪ Un graphe connexe de taille n admet au moins $n - 1$ arêtes : cela prouve $1 \Rightarrow 3$.

↪ Un graphe acyclique de taille n admet au plus $n - 1$ arêtes : cela prouve $2 \Rightarrow 4$.

↪ On en déduit qu'un arbre de taille n admet $n - 1$ arêtes d'où $0 \Rightarrow 1$ et $0 \Rightarrow 2$.

↪ Si 3 est vérifiée alors l'arbre est acyclique car sinon on pourrait enlever une des arête d'un cycle donc $3 \Rightarrow 0$.

↪ Si 4 est vérifiée alors l'arbre est connexe car sinon on pourrait ajouter une arête entre deux sommets non équivalents donc $4 \Rightarrow 0$.

Éléments de réponse pour l'exercice 15

La somme des degré vaut le double du nombre d'arêtes : $2n - 2$.

Si a est le nombre de sommets de degré 1 et $b = n - a$ le nombre de sommets de degré au moins 2 on a $2n - 2 \geq a + 2b = 2n - a$ d'où $a \geq 2$.

Si $a = 2$ alors l'inégalité est une égalité donc les autres sommets sont de degré 2. En suivant les sommets depuis un sommet de degré 1 on en déduit que G est linéaire.

Éléments de réponse pour l'exercice 16

Chaque fois que l'on ajoute un sommet t en visitant s on ajoute l'arête (s, t) . L'ensemble de ces arêtes permet de construire des chemins depuis s_0 vers tous les sommets accessibles, elles forment un graphe connexe. On ne revient jamais vers un sommet déjà inclus donc il n'y a pas de cycle simple.

Éléments de réponse pour l'exercice 17

```
let arbreProfondeur g s0 =
  let vus = make_vect (taille g) false in
  let arbre = ref [] in
  let rec visiter s =
    let appel t = if not vus.(t)
                  then arbre := (s,t)::!arbre;
                    visiter t in
    vus.(s) <- true;
    do_list appel (voisins g s) in
  visiter s0;
  !arbre;;
```

```
let arbreLargeur g s0 =
  let vus = make_vect (taille g) false in
  let attente = createQueue () in
  let arbre = ref [] in
  let traiter orig s = if not vus.(s)
                      then (enque s attente;
                            vus.(s) <- true;
                            arbre := (orig,s)::!arbre )in
  enque s0 attente;
  vus.(s0) <- true;
  while not (isEmptyQueue attente) do
    let s = first attente in
    deque attente;
    do_list (traiter s) (voisins g s) done;
  !arbre;;
```

Éléments de réponse pour l'exercice 18

```

let distance g s0 =
  let n = taille g in
  let distance = Array.make n (-1) in
  let attente = Queue.create() in
  let traiter k s = if distance.(s) = -1
                    then (Queue.add s attente;
                          distance.(s) <- k) in
  Queue.add s0 attente;
  distance.(s0) <- 0;
  while not (Queue.is_empty attente) do
    let s = Queue.take attente in
    let d = distance.(s) + 1 in
    List.iter (traiter d) (voisins g s) done;
  distance;;

```

Éléments de réponse pour l'exercice 19

```

let warshall g =
  let n = taille g in
  let dist = Array.make_matrix n n n in
  for i = 0 to (n-1) do
    dist.(i).(i) <- 0;
    List.iter (fun k -> dist.(i).(k) <- 1) (voisins g i) done;
  for k = 0 to (n-1) do
    for i = 0 to (n-1) do
      for j = 0 to (n-1) do
        dist.(i).(j) <- min dist.(i).(j) (dist.(i).(k) + dist.(k).(j))
      done
    done
  done;
  distance;;

```

La complexité est en $\mathcal{O}(n^3)$.

Éléments de réponse pour l'exercice 20

On commence par écrire une fonction de conversion de matrice : a est la valeur par défaut, b est la valeur pour les arêtes, c est la valeur pour la diagonale.

```

let convert g a b c =
  let n = taille g in
  let gg = Array.make_matrix n n a in
  for i = 0 to (n-1) do
    gg.(i).(i) <- c;
    List.iter (function k -> gg.(i).(k) <- b) voisins g i done;
  gg;;

```

Il reste alors à modifier l'exercice précédent en ajoutant la définition de la fonction. L'assemblage des deux liste ne doit pas répéter le passage par k d'où la décapitation de la seconde liste.

```

let cheminWarshall g =
  let n = vect_length g in
  let distance = convert g n 1 0 in
  let constr = convert g Sans Direct Direct in
  for k = 0 to (n-1) do
    for i = 0 to (n-1) do
      for j = 0 to (n-1) do
        if distance.(i).(k)+distance.(k).(j) < distance.(i).(j)
        then (distance.(i).(j) <- distance.(i).(k)+distance.(k).(j);
              constr.(i).(j) <- Milieu k) done done done;
    let rec chemin i j =
      match constr.(i).(j) with
      |Sans -> failwith "Sommets non connectes"
      |Direct -> [i; j]
      |Milieu k -> (chemin i k)@(List.tl (chemin k j))
    in chemin;;

```

Éléments de réponse pour l'exercice 21

On suppose que, dans un graphe orienté, tous les sommets ont degré sortant non nul. n est le nombre de sommets. Chaque sommet admet, au moins, un voisin.

On part d'un sommet s_0 : il admet un voisin s_1 .

On construit ainsi un chemin de $n + 1$ sommets en ajoutant, à chaque étape, un voisin du sommet précédent $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_n$.

Parmi ces $n + 1$ sommets, deux aux moins sont égaux donc le chemin contient un circuit.

La contraposée du résultat prouvé est le lemme.

Éléments de réponse pour l'exercice 22

- On suppose que G admet un ordre topologique.
S'il admettait un circuit $s_0 \rightarrow s_1 \rightarrow \dots \rightarrow s_{p-1} \rightarrow s_0$ alors on devrait avoir $\sigma(s_0) < \sigma(s_1) < \dots < \sigma(s_{p-1}) < \sigma(s_0)$ ce qui est impossible : G est sans circuit.
- On prouve le sens direct par récurrence sur la taille, n , de G .
Un graphe de taille 1 est sans circuit et admet un ordre topologique.
On suppose que les graphes de taille n sans circuit admettent un ordre topologique.
Si G est de taille $n + 1$ et est sans circuit, il admet un sommet sans degré sortant d'après le lemme. On le note s et on pose $S' = S \setminus \{s\}$.
Le graphe induit par S' est de taille n et n'admet pas de circuit.
D'après l'hypothèse de récurrence il admet un ordre topologique σ' de S' vers $\{0, 1, \dots, n - 1\}$. On prolonge σ' sur S par $\sigma(s) = n$.
 σ est un ordre topologique sur S car les seules arêtes de G qui ne sont pas dans le graphe induit sont de la forme (t, s) avec $t \in S'$ donc $\sigma(t) = \sigma'(t) \leq n - 1 < n = \sigma(s)$.
On a donc un ordre topologique pour tout graphe orienté sans circuit.

Éléments de réponse pour l'exercice 23

On considère une arête (s, t) du graphe.

- Si le parcours visite s avant t , alors comme t est voisin de s , il sera visité pendant l'appel de `visiter s` donc sera ajouté dans la liste `tri` avant s : on a bien s avant t .
- On suppose que le parcours visite t avant s .
Il n'existe pas de chemin dans le graphe reliant t à s car sinon on pourrait adjoindre l'arête (s, t) à ce chemin pour construire un circuit.
Dans ce cas l'exploration depuis t se termine avant d'avoir visité s et on a encore s avant t dans la liste.

La liste fournit bien un ordre topologique.

Éléments de réponse pour l'exercice 24

- ↪ S'il existe un circuit on considère le premier sommet appartenant à un cycle appelé lors du parcours complet : s_0 .
Pendant qu'il est dans l'état `EnCours` il va appeler tous ses voisins qui ne seront vus qu'après avoir visité tous leurs voisins et ainsi de suite. En particulier un cycle simple va être parcouru et va revenir à s_0 encore marqué `EnCours`.
- ↪ Inversement si, lors du traitement d'un sommet s , l'algorithme appelle s cela signifie qu'il existe un chemin de s à s : c'est un circuit.

```

let test_acyclique g =
  let n = taille g in
  let etat = Array.make n NonVu in
  let acyclique = ref true in
  let rec visiter s =
    match etat.(s) with
    |Vu -> ()
    |EnCours -> acyclique := false
    |NonVu -> begin etat.(s) <- EnCours;
                  List.iter visiter (voisins g s);
                  etat.(s) <- Vu end in
  for i = 0 to (n-1) do visiter i done;
  !acyclique;;

```

Éléments de réponse pour l'exercice 25

On considère les deux cas

1. Si le parcours visite s avant t , alors comme t est accessible depuis s , il sera visité pendant l'appel de `visiter s` donc sera ajouté dans la liste `tri` avant s : on a bien s avant t .
2. On suppose que le parcours visite t avant s .
Il n'existe pas de chemin dans le graphe reliant t à s donc l'exploration depuis t se termine avant d'avoir visité s et on a encore s avant t dans la liste.

Éléments de réponse pour l'exercice 26

On peut utiliser le type abstrait

```

let transpose g =
  let n = taille g in
  let gt = creerGraphe n in
  for i = 0 to (n-1) do
    List.iter (fun x -> ajouter gt x i) (voisins g i) done;
  gt;;

```

On peut aussi spécialiser dans le cas d'un tableau d'adjacences pour éviter la complexité de l'ajout qui commence par tester si l'arête est déjà présente.

```

let transpose g =
  let n = taille g in
  let gt = Array.make n [] in
  for i = 0 to (n-1) do
    List.iter (fun x -> gt.(x) <- i::gt.(i)) g.(i) done;
  gt;;

```

Éléments de réponse pour l'exercice 27

Un chemin de s à t dans G définit un chemin de t à s dans G^T .

Ainsi il existe un chemin de s à t et un chemin de t dans s dans G si et seulement il existe un chemin de t à s et un chemin de s dans t dans G^T . Les relations de sommets équivalents sont donc les mêmes dans G et dans G^T et les classes d'équivalence sont aussi les mêmes : ce sont les composantes connexes.

Éléments de réponse pour l'exercice 28

Soit s un point de G (et de G^T). On note C sa composante fortement connexe.

1. Tous les successeurs de s dans G^T qui ne sont pas dans C sont des points t tels qu'il existe un chemin de s vers t mais pas de chemin de t vers s dans G^T . Ce sont donc des points tels qu'il existe un chemin de t vers s mais pas de chemin de s vers t dans G .
D'après l'exercice **Ex. 25** on en déduit que t est placé avant s dans la liste ordre.
2. De même s ne peut être appelé dans le traitement d'un sommet u n'appartenant pas à C que si u est placé après s dans ordre.
3. Ainsi, lorsque s est traité, tous ses successeurs ont déjà été traités donc la fonction ne traite que des sommets de C .

Lors du traitement du premier point de C on ne traite que des points de C et on les traite tous car aucun n'a déjà été traité.

Éléments de réponse pour l'exercice 29

```
let composantesFC g =
  let gt = transpose g in
  let cfc = ref [] in
  let vus= Array.make (taille g) false in
  let rec visiter comp s = print_int s; print_string " : ";
    if not vus.(s)
    then (vus.(s) <- true; print_int s; print_string ", ";
      comp := s::(!comp);
      List.iter (visiter comp) (voisins gt s)) in
  let visitercfc s =
    if not vus.(s)
    then begin
      let comp = ref [] in
      visiter comp s;
      cfc := (!comp) :: (!cfc) end in
  List.iter visitercfc (tri_final g); !cfc;;
```

5 Graphes Valués

Éléments de réponse pour l'exercice 1

```
let creerGraphe n = Array.make_matrix n n Infini;;
```

```
let taille g = Array.length g;;
```

```
let existe s1 s2 g =  
  match g.(s1).(s2) with  
  |Infini-> false  
  |Poids _ -> true;;
```

```
let poids s1 s2 g =  
  match g.(s1).(s2) with  
  |Infini -> failwith "Il n'y a pas d'arc"  
  |Poids x -> x;;
```

```
let ajouter s1 s2 poids g =  
  g.(s1).(s2) <- (Poids poids);;
```

```
let retirer s1 s2 g =  
  g.(s1).(s2) <- Infini;;
```

```
let voisins s g =  
  let v = ref [] in  
  for i = 0 to (taille g - 1) do  
    match g.(s).(i) with  
    |Infini -> ()  
    |Poids k -> v := (i,k)::(!v) done;  
  !v;;
```

```

let aretes g =
  let a = ref [] in
  for i = 0 to (taille g - 1) do
    for j = 0 to (taille g - 1) do
      match g.(i).(j) with
      |Infini -> ()
      |Poids k -> a := (i,j,k)::(!a) done done;
  !a;;

```

Éléments de réponse pour l'exercice 2

On utilise les fonctions du module List :

↪ la fonction de test dans une liste

```
List.exists test liste : ('a -> bool) -> 'a list -> bool
```

qui renvoie true si et seulement si il existe un terme de la liste qui vérifie le test, il existe aussi la fonction List.for_all test liste

↪ la fonction qui renvoie l'élément associé à un autre dans une liste de couples

```
List.assoc a liste : 'a -> ('a * 'b) list -> 'b
```

Il existe une fonction de test adapté :

```
List.mem_assoc : 'a -> ('a * 'b) list -> bool
```

```
let creerGraphe n = Array.make n [];
```

```
let taille g = Array.length g;;
```

```
let existe s1 s2 g =
  List.exists (fun (s,_) -> s=s2) g.(s1);;
```

```
let poids s1 s2 g =
  List.assoc s2 g.(s1);;
```

```
let ajouter s1 s2 poids g =
  if not existe s1 s2 g
  then g.(s1) <- (s2,poids)::(g.(s1));;
```

```

let retirer s1 s2 g =
  let rec moins s liste =
    match liste with
    | [] -> []
    |(e,w)::q when e = s -> q
    |t::q -> t::(moins s q) in
  let liste = moins s2 g.(s1) in
  g.(s1) <- liste;;

```

```

let voisins g s = g.(s);;

```

```

let aretes g =
  let a = ref [] in
  for i = 0 to (taille g -1) do
    let ar = map (fun (j,k) -> (i,j,k)) g.(i) in
    a := ar@(!a) done;
  !a;;

```

Éléments de réponse pour l'exercice 3

On considère les sommets parcourus par c .

$$a = s_0 \xrightarrow{w_1} s_1 \xrightarrow{w_2} \dots \xrightarrow{w_i} s_i \xrightarrow{w_{i+1}} \dots \xrightarrow{w_j} s_j \xrightarrow{w_{j+1}} \dots \xrightarrow{w_m} s_m = b$$

S'il existe deux sommets égaux dans le chemin $s_i = s_j$ on peut retirer la portion de chemin entre s_i et s_j et obtenir un chemin de a à b de poids strictement inférieur (de $w_{i+1} + \dots + w_j$).

Après un nombre fini de telles opérations on aboutit à un chemin sans boucle de poids inférieur.

Les chemins sans boucle passent par des sommets distincts donc sont de longueur $n - 1$ au plus donc sont en nombre fini ; on peut donc considérer un chemin de poids minimum, c_0 .

D'après la question qui précède c_0 est de poids minimum.

Éléments de réponse pour l'exercice 4

On a $p_k(i, k) = \min\{p_{k-1}(i, k), p_{k-1}(i, k) + p_{k-1}(k, k)\} = p_{k-1}(i, k)$ car $p_{k-1}(k, k) = 0$.

Les formules de passages de P_{k-1} à P_k peuvent donc utiliser indistinctement les termes $p_{k-1}(i, k)$ ou $p_k(i, k)$; on peut donc travailler sur les termes de la matrice en utilisant les valeurs des $p_{i,k}$ qu'ils aient été transformés ou non.

Éléments de réponse pour l'exercice 5

Un chemin de poids minimum de s à s' concaténé avec un chemin de poids minimum de s' à s'' donne un chemin de s à s'' de poids $d(s, s') + d(s', s'')$. Ce poids est minoré par le poids minimum $d(s, s'')$.

Éléments de réponse pour l'exercice 6

```
let plus a1 a2 =
  match (a1,a2) with
  |(Infini,_) -> Infini
  |(_,Infini) -> Infini
  |(Poids p1,Poids p2) -> Poids (p1+p2);;

let petit a1 a2 =
  match (a1,a2) with
  |(Infini,_) -> a2
  |(_,Infini) -> a1
  |(Poids p1,Poids p2) -> Poids (min p1 p2);;
```

Éléments de réponse pour l'exercice 7

On a besoin, pour savoir si k est introduit, de déterminer si le chemin composé a un poids strictement inférieur :

```
let plusPetitStrict a1 a2 =
  match (a1,a2) with
  |(Infini,_) -> false
  |(_,Infini) -> true (* Le cas de 2 infinis a été traité au dessus *)
  |(Poids p1,Poids p2) -> p1 < p2;;
```

On modifie alors le programme

```

let cheminFloyd g =
  let n = taille g in
  let p = Array.make_matrix n n Infini in
  let ch = Array.make_matrix n n Sans in
  for i = 0 to (n-1) do
    List.iter (fun (j,k) -> p.(i).(j) <- Poids k) (voisins g i);
    List.iter (fun (j,k) -> ch.(i).(j) <- Direct) (voisins g i);
    p.(i).(i) <- Poids 0;
    ch.(i).(i) <- Direct done;
  for k = 0 to (n-1) do
    for i = 0 to (n-1) do
      for j = 0 to (n-1) do
        let p1 = p.(i).(j) in
        let p2 = plus p.(i).(k) p.(k).(j) in
        if plusPetitStrict p2 p1
        then (p.(i).(j) <- p2; ch.(i).(j) <- Milieu (k))
      done
    done
  done;
let rec chemin i j =
  match ch.(i).(j) with
  |Sans -> failwith "Sommets non connectes"
  |Direct -> [i; j]
  |Milieu k -> (chemin i k)@(List.tl (chemin k j)) in
chemin;;

```

6 Compléments : Flots

Éléments de réponse pour l'exercice 1

Dans ces conditions on a $\varphi(i, j) \leq c(i, j) = 0$ et $\varphi(i, j) = -\varphi(j, i) \geq -c(j, i) = 0$.

Éléments de réponse pour l'exercice 2

On a $Z = \sum_{i \in T} \sum_{j \in T} \varphi(i, j) = \sum_{i \in T} \sum_{j \in T} -\varphi(j, i) = - \sum_{j \in T} \sum_{i \in T} \varphi(j, i) = -Z$ donc $Z = 0$.

Ainsi $0 = \sum_{i \in S} \sum_{j \in S} \varphi(i, j) = \sum_{j \in S} \varphi(s, j) + \sum_{i \in S \setminus \{s, p\}} \sum_{j \in S} \varphi(i, j) + \sum_{j \in S} \varphi(p, j)$

d'où le résultat car on obtient $0 = |\varphi| + \sum_{i \in S \setminus \{s, p\}} 0 + \sum_{j \in S} -\varphi(j, p)$.

Éléments de réponse pour l'exercice 3

↪ Comme le chemin est simple il ne peut pas contenir une arête (i, j) en même temps que l'arête (j, i) : φ est bien définie.

↪ On a $\varphi(s_{i-1}, s_i) = \omega \leq w(s_{i-1}, s_i) = c(s_{i-1}, s_i)$,

$$\varphi(s_i, s_{i-1}) = -\omega \leq 0 \leq c(s_{i-1}, s_i)$$

et $\varphi(i, j) = 0 = c(i, j)$ pour les autres couples

La majoration est vérifiée.

↪ La symétrie est valide par construction.

↪ Si s n'est pas un point du chemin $\sum_{j \in S} \varphi(i, j) = \sum_{j \in S} 0 = 0$

si $i = s_k$ est distinct de s et p alors on a $1 \leq k \leq r - 1$ et

$$\sum_{j \in S} \varphi(s_k, j) = \varphi(s_k, s_{k-1}) + \varphi(s_k, s_{k+1}) = -\omega + \omega = 0.$$

On a $|\varphi| = w(s, s_1) > 0$.

Éléments de réponse pour l'exercice 4

On note c_φ la capacité de R_φ , on a $\varphi(i, j) + \psi(i, j) \leq \varphi(i, j) + c_\varphi(i, j)$.

Si $(i, j) \in A_\varphi$ alors $c_\varphi(i, j) = 0$ donc $\varphi(i, j) + \psi(i, j) = \varphi(i, j) \leq c(i, j)$.

Si $(i, j) \in A_\psi$ alors $c_\varphi(i, j) = w_\varphi(i, j) = c(i, j) - \varphi(i, j)$ d'où $\varphi(i, j) + \psi(i, j) \leq c(i, j)$.

Les deux autres propriétés de flots et l'additivité sont immédiates.

Éléments de réponse pour l'exercice 5

S'il existe un chemin améliorant alors la construction de l'exercice 3 permet de construire un flot, ψ , sur R_φ avec $|\psi| > 0$. L'exercice 4 permet alors de construire un nouveau flot, $\varphi + \psi$ de valeur $|\varphi| + |\psi| > |\varphi|$ donc φ n'est pas maximal.

Le résultat en découle par contraposition.

Éléments de réponse pour l'exercice 6

$$\begin{aligned} \sum_{i \in S_s} \sum_{j \in S_p} \varphi(i, j) &= \sum_{i \in S_s} \sum_{j \in S \setminus S_s} \varphi(i, j) = \sum_{i \in S_s} \left(\sum_{j \in S} \varphi(i, j) - \sum_{j \in S_s} \varphi(i, j) \right) \\ &= \sum_{i \in S_s} \sum_{j \in S} \varphi(i, j) - \sum_{i \in S_s} \sum_{j \in S_s} \varphi(i, j) \\ &= \sum_{j \in S} \varphi(s, j) + \sum_{i \in S_s \setminus \{s\}} \sum_{j \in S} \varphi(i, j) - 0 = |\varphi| + \sum_{i \in S_s \setminus \{s\}} 0 = |\varphi| \end{aligned}$$

On a alors $|\varphi| = \sum_{i \in S_s} \sum_{j \in S_p} \varphi(i, j) \geq \sum_{i \in S_s} \sum_{j \in S_p} c(i, j) = c(A, B)$.

Éléments de réponse pour l'exercice 7

Si une coupure est minimale pour un flot cela signifie que la valeur du flot a atteint un majorant, cette valeur est donc maximale.

Éléments de réponse pour l'exercice 8

On suppose qu'il n'existe pas de chemin améliorant dans R_φ .

Si A est l'ensemble des sommets accessibles depuis s dans R_φ alors p n'appartient pas à A donc (A, B) est une coupure de R associée à φ .

Pour $i \in A$ et $j \in B$ (i, j) n'est pas une arête de R_φ car sinon j serait accessible depuis i donc depuis s , ce qui est impossible : $(i, j) \notin A_\varphi$. D'après la définition de A_φ cela implique qu'on n'a pas $\varphi(i, j) < c(i, j)$ donc, en raison de la majoration, $\varphi(i, j) = c(i, j)$.

On a alors, d'après l'exercice 6, $|\varphi| = \sum_{i \in A} \sum_{j \in B} \varphi(i, j) = \sum_{i \in A} \sum_{j \in B} c(i, j) = c(A, B)$

donc (A, B) est une coupure minimale. On a prouvé $3 \Rightarrow 2$

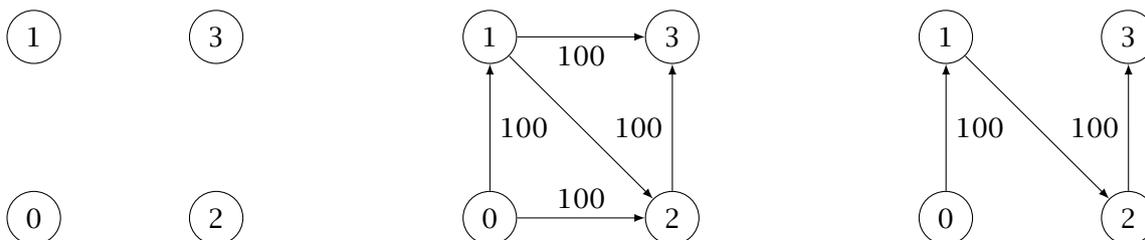
Éléments de réponse pour l'exercice 9

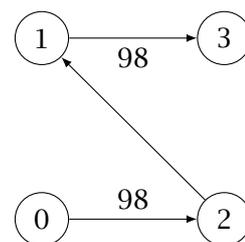
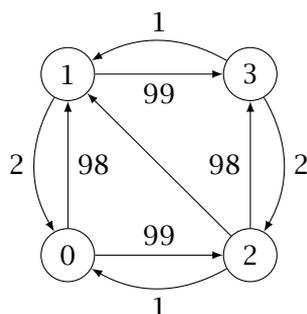
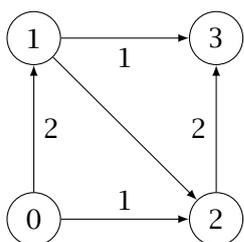
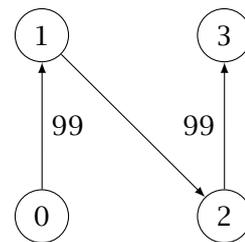
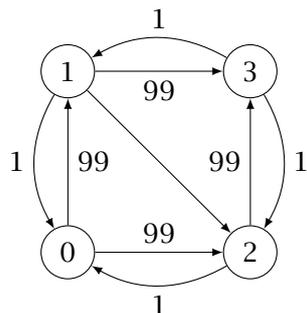
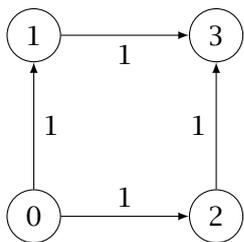
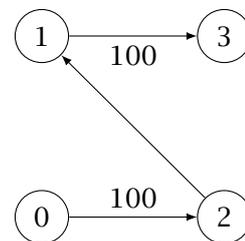
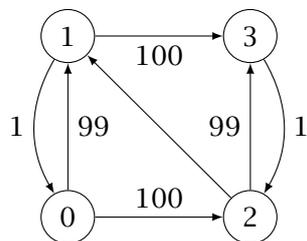
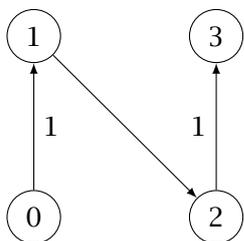
Dans le cas de poids entiers, la valeur du flot augment de 1 au moins à chaque itération.

Or la valeur d'un algorithme est majoré par $C(A, B)$ donc le nombre d'itérations est majoré par cet entier. Pour $A = \{s\}$ on trouve la majoration demandée.

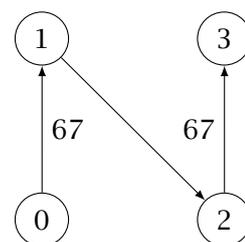
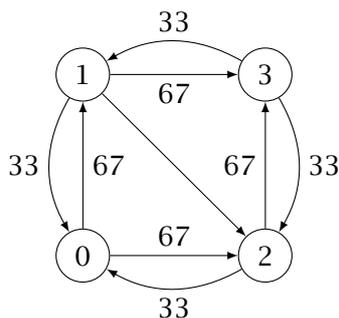
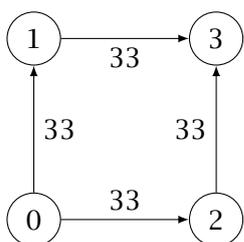
Si on passe par la diagonale dans $R_{2,\varphi}$, comme sa capacité ne peut être que 1 ou 0 on ne peut augmenter la valeur que de 1 à chaque itération : il faut 200 itérations.

Les représentations ci-dessous montre bien qu'on peut toujours passer par la diagonale. On présente, à chaque ligne, le flot de départ, le réseau résiduel et le chemin améliorant choisi.

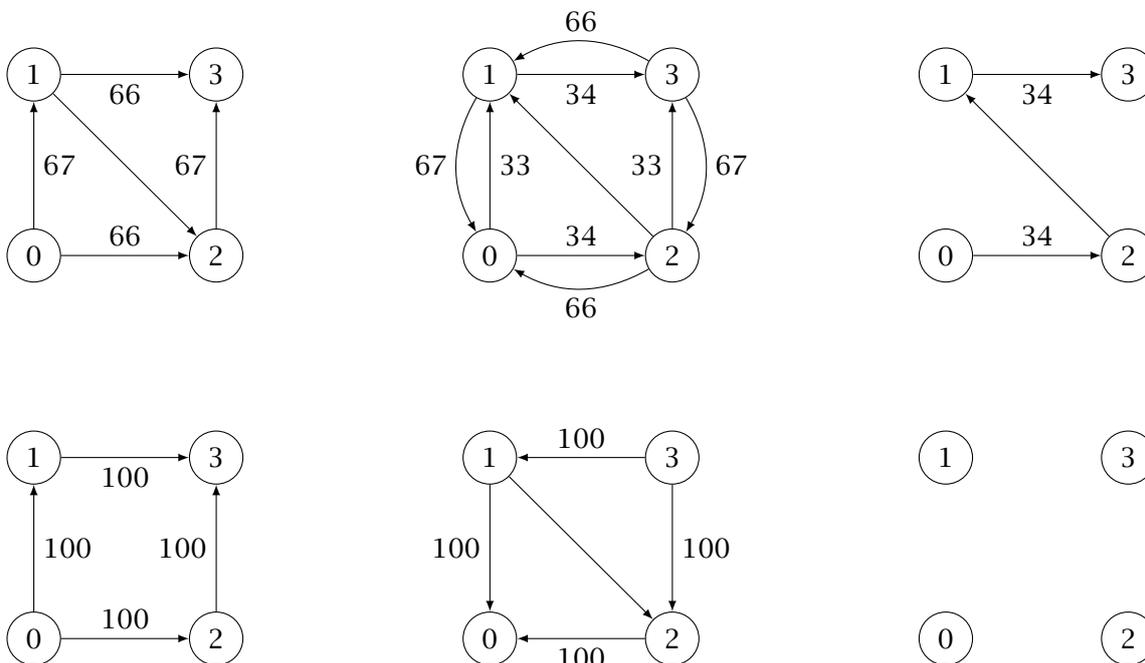




...



...



Éléments de réponse pour l'exercice 10

Si (i, j) est critique alors $(i, j) \in A_\varphi$ donc $\varphi(i, j) < c(i, j)$
 et aussi $(i, j) \notin A_{\varphi'}$ d'où $\varphi'(i, j) = c(i, j)$.

Alors $\psi(i, j) = \varphi'(i, j) - \varphi(i, j) = c(i, j) - \varphi(i, j) > 0$
 ce qui implique que (i, j) est une arête de c .

De plus $\omega = \psi(i, j) = c(i, j) - \varphi(i, j) = c_\varphi(i, j)$ le poids de (i, j) est bien ω .

Inversement, par construction, toute arête de poids ω de c est transformée en une arête de poids nul dans $R_{\varphi'}$ donc disparaît.

Éléments de réponse pour l'exercice 11

Si (i, j) est émergente alors $(i, j) \notin A_\varphi$ donc $\varphi(i, j) = c(i, j)$
 et aussi $(i, j) \in A_{\varphi'}$ d'où $\varphi'(i, j) < c(i, j)$.

Alors $\psi(i, j) = \varphi'(i, j) - \varphi(i, j) = \varphi'(i, j) - c(i, j) < 0$ d'où $\psi(j, i) > 0$: (j, i) est bien une arête de c .

Inversement si (j, i) est une arête de c alors $\psi(i, j) < 0$ donc, si on a de plus $\varphi(i, j) = c(i, j)$ car (i, j) n'appartient pas à A_φ on a $\varphi'(i, j) = \psi(i, j) + \varphi(i, j) < \varphi(i, j) = c(i, j)$ et (i, j) appartient à $A_{\varphi'}$.

Comme le chemin améliorant est simple il ne peut pas repasser par s donc $i \neq s$.

De même, comme il finit par p il ne peut pas être passé par p donc $j \neq p$.

Éléments de réponse pour l'exercice 12

Soient $t \in S_{\varphi'}$ et γ un chemin de de longueur minimale de s vers t dans $R_{\varphi'}$.

1. Si toutes les arêtes de γ sont aussi des arêtes dans R_φ alors γ est aussi un chemin dans R_φ donc t est accessible depuis s dans R_φ .
2. On suppose que γ n'est pas dans R_φ . Il contient donc au moins une arête émergente. On considère la dernière arête émergente de γ , (i, j) .
On a vu que (j, i) est une arête d'un chemin minimal de s à p dans R_φ .
En particulier j est accessible depuis s dans R_φ donc, comme toutes les arêtes entre j et t appartiennent à A_φ , t est accessible depuis s dans R_φ .

Dans les deux cas on a $t \in S_\varphi$ d'où l'inclusion.

Éléments de réponse pour l'exercice 13

Un chemin minimal de s à t est aussi un chemin minimal dans $R_{\varphi'}$ de s à tout point parcouru, en particulier de s vers i et j .

On en déduit qu'on a $d_{\varphi'}(j) = d_{\varphi'}(i) + 1$.

On a vu que, comme (i, j) est émergent, (j, i) est une arête de c , chemin minimal de s vers p dans R_φ . On en déduit, de même que $d_\varphi(i) = d_\varphi(j) + 1$. On obtient l'égalité demandée en additionnant les deux égalités trouvées.

On montre alors que si un chemin minimal de s à t dans $R_{\varphi'}$ contient q arêtes émergentes, on a $d_{\varphi'}(t) \geq d_\varphi(t) + 2q$. On le prouve par récurrence sur q .

→ Pour $q = 0$, le chemin minimal de s à t dans $R_{\varphi'}$ est aussi un chemin dans R_φ donc sa longueur majore $d_\varphi(t)$ c'est-à-dire $d_{\varphi'}(t) \geq d_\varphi(t)$.

→ On suppose la propriété valide pour $q \geq 0$.

On suppose qu'un chemin minimal de s à t dans $R_{\varphi'}$ contient $q + 1$ arêtes émergentes.

On considère la dernière de ces arêtes émergentes : (i, j) .

On note p la longueur (éventuellement nulle) du chemin entre j et t .

La restriction du chemin de s à i est minimale et contient q arêtes émergentes.

D'après l'hypothèse de récurrence on a $d_{\varphi'}(i) \geq d_\varphi(i) + 2q$.

La propriété ci-dessus montre qu'on a

$$d_{\varphi'}(j) = d_\varphi(j) + d_{\varphi'}(i) - d_\varphi(i) + 2 \geq d_\varphi(j) + 2(q + 1).$$

Comme le chemin de j à t est minimal on a $d_{\varphi'}(t) = d_{\varphi'}(j) + p$.

Ce chemin de j à t est aussi un chemin dans R_φ donc $d_\varphi(t) \leq d_\varphi(j) + p$.

Ainsi $d_{\varphi'}(t) = d_{\varphi'}(j) + p \geq d_\varphi(j) + 2(q + 1) + p \geq d_\varphi(t) + p$.

La propriété est vérifiée pour $q + 1$.

On a prouvé la récurrence. On en déduit $d_{\varphi'}(t) \geq d_\varphi(t)$.

Éléments de réponse pour l'exercice 14

Lorsqu'une arête (i, j) est critique, elle disparaît lors du calcul du nouveau flot, de φ_1 à φ'_1 . On a vu qu'alors elle est une arête du chemin minimal donc $d_{\varphi_1}(j) = d_{\varphi_1}(i) + 1$.

Pour qu'elle puisse redevenir critique il faut qu'elle émerge entre-temps.

Lors de l'émergence lors du calcul du nouveau flot de φ_2 à φ_2' on a (j, i) qui est une arête du chemin minimal donc $d_{\varphi_2}(i) = d_{\varphi_2}(j) + 1$.

On a $d_{\varphi_2}(j) \geq d_{\varphi_1}(j)$ donc $d_{\varphi_2}(i) \geq d_{\varphi_1}(i) + 1$.

La distance de s à i augmente donc d'au moins 2 à chaque cycle critique-émergence.

Or la distance minimale de i est 1 ($i \neq s$) et sa distance maximale est $|S| - 1$. Donc il ne peut y avoir plus de $(|S| - 1 - 1)/2$ cycles.

On peut ajouter un dernier passage critique : le nombre de tels passages est donc majoré par $|S|/2$.

Les seules arêtes possibles dans les réseaux R_φ sont les arêtes de A ou leurs inverses : il y en a donc $2|A|$. Il ne peut ainsi advenir que $2|A| \times |S|/2$ arêtes critiques.

Or chaque itération engendre au moins une arête critique (celle dont le poids est ω) : le nombre d'itérations est majoré par $|A| \times |S|$.

La complexité du parcours en profondeur dans le cas d'une représentation matricielle est un $\mathcal{O}(|S|^2)$ donc la complexité de l'algorithme est un $\mathcal{O}(|S|^3 \cdot |A|)$.

Comme $|A| \leq |S|^2$ la complexité est aussi un $\mathcal{O}(|S|^5)$.

Éléments de réponse pour l'exercice 15

```
let voisins g s =
  let v = ref [] in
  let n = Array.length g in
  for i = 0 to (n - 1) do
    if g.(s).(i) > 0
    then v := i::(!v) done;
  !v;;
```

```
let rechLarg g s0 =
  let n = Array.length g in
  let peres = Array.make n (-1) in
  let attente = Queue.create () in
  let traiter pere s = if peres.(s) = -1
    then (Queue.add s attente;
          peres.(s) <- pere) in
  Queue.add s0 attente;
  peres.(s0) <- s0;
  while not (Queue.is_Empty attente) do
    let s = Queue.take attente in
    do_list (traiter s) (voisins g s) done;
  peres;;
```

```

let cheminMin g a b =
  let peres = rechLarg g a in
  let rec chemin k fait =
    if peres.(k) = -1
    then []
    else if peres.(k) = k
         then k::fait
         else chemin peres.(k) (k::fait) in
  chemin b [];;

```

Éléments de réponse pour l'exercice 16

```

let rec poidsMin chemin g =
  match chemin with
  |[s; t] -> g.(s).(t)
  |s::t::reste -> min (poidsMin (t::reste) g) g.(s).(t)
  |_ -> 0;;

```

```

let modif chemin g phi=
  let w = poidsMin chemin g in
  let rec aux chem =
    match chem with
    |s::t::reste -> g.(s).(t) <- g.(s).(t) - w;
                  g.(t).(s) <- g.(t).(s) + w;
                  phi.(s).(t) <- phi.(s).(t) + w;
                  phi.(t).(s) <- phi.(t).(s) - w;
                  aux (t::reste)

    |_ -> () in
  aux chemin;;

```

Éléments de réponse pour l'exercice 17

```
let flotMax g s p =
  let n = vect_length g in
  let phi = make_matrix n n 0 in
  let capa = make_matrix n n 0 in
  for i = 0 to (n-1) do
    for j = 0 to (n-1) do
      capa.(i).(j) <- poids g i j done done;
  let chem = ref (cheminMin capa s p) in
  while !chem <> [] do
    let w = poidsMin !chem capa in
    modif !chem capa phi;
    chem := cheminMin capa s p done;
  phi;;
```

7 Langages rationnels

Éléments de réponse pour l'exercice 1

On note $T(n, p)$ le nombre de comparaisons effectuées dans la recherche d'un mot de longueur p dans un texte de longueur n .

On effectue $n - p + 1$ comparaisons de mots de longueur p .

Ces comparaisons demandent au plus p comparaisons de caractères

donc $T(n, p) \leq (n - p + 1) \cdot p = \mathcal{O}(np)$.

Le pire des cas advient avec `texte = "aaaaaaaaaaaaaaaaaaaaa"` et `motif = "aaaaaa"`.

Dans ce cas on doit attendre la dernière comparaison de caractère pour voir que le motif apparaît pour chaque position : le nombre de comparaison est égal à $(n - p + 1) \cdot p$.

Éléments de réponse pour l'exercice 2

$L_k \cap L_j = \emptyset$ si $k - j$ n'est pas un multiple de 3, $L_k \cap L_j = L_p$ avec $p = \min(k, j)$ sinon.

$L_k \cdot L_j = L_{k+j}$

Éléments de réponse pour l'exercice 3

On note $u = u_1 u_2 \dots u_n$, $v = v_1 v_2 \dots v_p$ et $w = w_1 w_2 \dots w_q$.

Les deux produits $(u.v).w$ et $u.(v.w)$ s'écrivent $t_1 t_2 \dots t_{n+p+q}$ avec $t_i = u_i$ pour $1 \leq i \leq n$, $t_i = v_{i-n}$ pour $n + 1 \leq i \leq n + p$ et $t_i = w_{i-n-p}$ pour $n + p + 1 \leq i \leq n + p + q$.

Les deux produits de langages sont égaux à $\{w_1.w_2.w_3 ; w_1 \in L_1, w_2 \in L_3, w_3 \in L_3\}$.

Éléments de réponse pour l'exercice 4

↪ Si $v < u$ alors $u = v.w$ donc $|u| = |v| + |w| \geq |v|$ et les lettres de v sont les premières lettres de u . Inversement si $|v| \leq |u|$ et $v_i = u_i$ pour $1 \leq i \leq |v|$ alors $u = v.w$ en posant $w = u_{|v|+1} \dots u_{|u|}$.

↪ $u = u.\varepsilon$ donc $u < u$ pour tout mot u .

Si $u < v$ et $v < u$ alors $|u| \leq |v|$ et $|v| \leq |u|$ donc $|u| = |v|$. De plus $u_i = v_i$ pour $1 \leq i \leq |u| = |v|$ donc $u = v$.

Si $u < v$ et $v < w$ alors $v = u.u'$ et $w = v.v'$ donc $w = u.(u'.v') : u < w$.

Éléments de réponse pour l'exercice 5

On prouve, par récurrence sur p , que $w^n . w^p = w^{n+p}$ en utilisant l'associativité.

On a alors $w^p . w^n = w^{p+n} = w^{n+p} = w^n . w^p$.

Éléments de réponse pour l'exercice 6

1. L'ensemble des mots de longueur n au plus est fini donc rationnel.
2. L'ensemble des mots de longueur n exactement, \mathcal{A}^n , est fini donc rationnel. L'ensemble des mots de longueur n au moins est le produit des langages rationnels \mathcal{A}^n et \mathcal{A}^* donc est rationnel
3. L'ensemble des mots qui commencent et qui finissent par la même lettre est l'union des $\{x\} . \mathcal{A}^* . \{x\}$ pour x décrivant Σ . Il est rationnel.

Éléments de réponse pour l'exercice 7

$$\sqrt{L_{2p}} = L_p, \sqrt{L_{2p+1}} = L_{p+2}.$$

Si $\epsilon \in L$ alors $\epsilon.\epsilon = \epsilon \in L$ donc $\epsilon \in \sqrt{L}$.

Si $\epsilon \in \sqrt{L}$ alors $\epsilon = \epsilon.\epsilon \in L$.

Éléments de réponse pour l'exercice 8

- $\hookrightarrow L_1 = \{ab, baa, abba, aabaa\} : ab.(baa)^3 = abba.ab.aabaa, L_1$ n'est pas un code

$\hookrightarrow L_2 = \{b, ab, baa, abaa, aaaa\} :$
on suppose que $x_1.x_2.\dots.x_p = y_1.y_2.\dots.y_q$ avec $x_1 \neq y_1$ (sinon on simplifie).
Si, par exemple, $|x_1| \leq |y_1|$ on voit que les seules possibilités sont $x_1 = b$ et $y_1 = baa$ ou $x_1 = ab$ et $y_1 = abaa$ ce qui impose que x_2 commence par aa donc $x_2 = aaaa$.
 y_2 doit alors commencer par aa donc $y_2 = aaaa$ et on voit que tous les termes restant sont $aaaa$ sans jamais pouvoir égaliser les longueurs. L_2 est un code

$\hookrightarrow L_3 = \{aa, ab, aab, bba\} :$
on suppose que $x_1.x_2.\dots.x_p = y_1.y_2.\dots.y_q$ avec $x_1 \neq y_1$ (sinon on simplifie).
Si, par exemple, $|x_1| \leq |y_1|$ on voit que $x_1 = aa$ et $y_1 = aab$ donc $x_2 = bba$ et y_2 devrait commencer par ba ce qui est impossible. L_3 est un code

$\hookrightarrow L_4 = \{a, ba, bba, baab\} : ba.a.bba = baab.ba, L_4$ n'est pas un code
- $u^n = u^p$ impose $n|u| = p|u|$ donc $n = p$ si $|u| \neq 0$. Ainsi $\{u\}$ est un code si $u \neq \epsilon$.
Par contre $\epsilon = \epsilon.\epsilon$ donc $\{\epsilon\}$ n'est pas un code.
- Si $u.v = v.u$ alors on a 2 décompositions d'un même mot : $\{u, v\}$ n'est pas un code.
On suppose que $\{u, v\}$ n'est pas un code. On peut écrire $x_1.x_2.\dots.x_p = y_1.y_2.\dots.y_q$ avec $x_1 \neq y_1$.
Par exemple $x_1 = u$ et $y_1 = v$. Ainsi u et v sont préfixes d'un même mot donc, par exemple, u est préfixe de v : on pose $v = u.v'$ on a $v' \neq \epsilon$ car $u \neq v$.
Si $v' = u$ alors $v = u^2$ donc $u.v = v.u$.
Si $u \neq v'$ on est alors ramené, en remplaçant v par $u.v'$ à un mot dans $\{u, v'\}$ qui admet deux écritures distinctes avec u et v' distincts dans $\{u, v'\}^+$.
On peut alors conclure par récurrence sur $|u| + |v|$.
En effet On a $|u| + |v'| < |u| + |v|$ donc $u.v' = v'.u$ d'où $u.v = u.u.v' = u.v'.u = v.u$.
- Si L n'est pas un code alors on peut écrire $x_1.x_2.\dots.x_p = y_1.y_2.\dots.y_q$ avec $x_1 \neq y_1$ (sinon on simplifie). Si, par exemple, $|x_1| \leq |y_1|$ on voit que x_1 est un préfixe de y_1 . Ainsi, si aucun mot de L n'est préfixe d'un autre mot de L , alors L est un code.

Éléments de réponse pour l'exercice 9

- On a $B \subset A.L \cup B = L$. Alors $A.B \subset A.L \subset A.L \cup B = L$.
Par récurrence, si $A^n.B \subset L$ alors $A^{n+1}.B = A.(A^n.B) \subset A.L \subset A.L \cup B = L$.
Ainsi $A^n.B \subset L$ pour tout n donc $A^*.B = \bigcup_{n \in \mathbb{N}} A^n.B \subset L$.
- On suppose que $\epsilon \notin A$ et que $L \neq A^*.B$. On a donc, comme $A^*.B \subset L, L \setminus A^*.B \neq \emptyset$.
Soit u un mot de longueur minimale appartenant à $L \setminus A^*.B$.

$u \in L = A.L \cup B$ donc

→ soit $u \in B \subset A^*.B$ ce qui est impossible

→ soit $u \in A.L$ donc $u = u_1u_2$ avec $u_1 \in A$ et $u_2 \in L$. On a $u_1 \neq \epsilon$ car $\epsilon \notin A$ donc $|u_1| > 0$ puis $|u_2| < |u|$. On a ainsi $u \in L$ de longueur strictement inférieure à celle de $|u|$ donc $u_2 \in A^*.B$ puis $u = u_1.u_2 \in A.(A^*.B) \subset A^*.B$ ce qui est impossible aussi.

On doit donc avoir $L \subset A^*.B$ d'où l'égalité.

3. On a $A^0.L = L \subset L$. Si $A^n.L \subset L$ alors $A^{n+1}.L = A.(A^n.L) \subset AL \subset L$ donc, par récurrence sur n on a $A^n.L \subset L$ pour tout n puis $A^*.L \subset L$.

L'inclusion inverse est évidente donc $L = A^*.L$ avec $B \subset A.L \cup B = L$.

On suppose maintenant que $\epsilon \in A$; soit $C \subset \Sigma^*$ tel que $B \subset C$.

On veut montrer que $L = A^*.C$ vérifie $A.L \cup B = L$.

On a $A.(A^*.C) = A^+.C \subset A^*.C$ et, comme $\epsilon \in A$, $A^*.C = \{\epsilon\}.(A^*.C) \subset A.(A^*.C)$ donc $A^*.C = A.(A^*.C)$. De plus $B \subset C \subset A^*.C$ donc

$A.L \cup B = A.(A^*.C) \cup B = A^*.C \cup B = A^*.C = L : A^*.C$ est bien solution.

4. Si un mot contient un nombre pair de b alors il peut être ϵ ou est de la forme $a.u$ avec u contenant un nombre pair de b ou est de la forme $b.u$ avec u contenant un nombre impair de b . Ainsi $L_1 = \{\epsilon\} \cup \{a\}.L_1 \cup \{b\}.L_2$.

De même $L_2 = \{a\}.L_2 \cup \{b\}.L_1$.

Comme $\epsilon \notin \{b\}$ la dernière équation impose $L_2 = \{a\}^*.\{b\}.L_1$ d'où

$L_1 = \{\epsilon\} \cup \{a\}.L_1 \cup \{b\}.\{a\}^*.\{b\}.L_1 = (\{a\} \cup \{b\}.\{a\}^*.\{b\}).L_1 \cup \{\epsilon\}$.

ϵ n'appartient pas à $\{a\} \cup \{b\}.\{a\}^*.\{b\}$ donc on doit avoir $L_1 = (\{a\} \cup \{b\}.\{a\}^*.\{b\})^*.\{\epsilon\} = (\{a\} \cup \{b\}.\{a\}^*.\{b\})^*$ puis $L_2 = \{a\}^*.\{b\}.\{a\} \cup \{b\}.\{a\}^*.\{b\})^*$

qui transforment On va définir par induction structurelle des transformations de langages f_1, f_2, f_3, f_4 et f_5 pour chaque question.

Éléments de réponse pour l'exercice 10

- $f_1(\emptyset) = \emptyset,$
 $f_1(\{\epsilon\}) = \{\epsilon\},$
 $f_1(\{x\}) = \{\epsilon, x\},$
 $f_1(L_1 \cup L_2) = f_1(L_1) \cup f_1(L_2),$
 $f_1(L_1.L_2) = f_1(L_1) \cup L_1.f_1(L_2),$
 $f_1(L^*) = L^*.f_1(L).$
- $f_2(\emptyset) = \emptyset,$
 $f_2(\{\epsilon\}) = \{\epsilon\},$
 $f_2(\{x\}) = \{x\}$ si $x \neq a$ et $f_2(\{a\}) = \emptyset,$
 $f_2(L_1 \cup L_2) = f_2(L_1) \cup f_2(L_2),$
 $f_2(L_1.L_2) = f_2(L_1).f_2(L_2),$
 $f_2(L^*) = (f_2(L))^*.$
- $f_3(\emptyset) = \emptyset,$
 $f_3(\{\epsilon\}) = \emptyset,$

$$f_3(\{x\}) = \emptyset \text{ si } x \neq a \text{ et } f_3(\{a\}) = \{a\},$$

$$f_3(L_1 \cup L_2) = f_2(L_1) \cup f_3(L_2),$$

$$f_3(L_1.L_2) = f_3(L_1).L_2 \cup L_1.f_3(L_2),$$

$$f_3(L^*) = L^*.f_3(L).L^*.$$

$$4. f_4(\emptyset) = \emptyset,$$

$$f_4(\{\epsilon\}) = \emptyset,$$

$$f_4(\{x\}) = \emptyset \text{ si } x \neq a \text{ et } f_4(\{a\}) = \{\epsilon\},$$

$$f_4(L_1 \cup L_2) = f_4(L_1) \cup f_4(L_2),$$

$$f_4(L_1.L_2) = f_4(L_1).L_2 \cup f_2(L_1).f_4(L_2),$$

$$f_4(L^*) = (f_2(L))^*.f_4(L).L^*.$$

$$5. f_5(\emptyset) = \emptyset,$$

$$f_5(\{\epsilon\}) = \emptyset,$$

$$f_5(\{x\}) = \emptyset \text{ si } x \neq a \text{ et } f_5(\{a\}) = \{\epsilon\},$$

$$f_5(L_1 \cup L_2) = f_5(L_1) \cup f_5(L_2),$$

$$f_5(L_1.L_2) = f_5(L_1).L_2 \cup L_1.f_5(L_2),$$

$$f_5(L^*) = L^*.f_5(L).L^*.$$

Éléments de réponse pour l'exercice 11

La démonstration se fait par induction structurelle. La propriété est immédiate pour les langages élémentaires.

Dans le tableau on étudie les cas d'expressions régulières qui dénotent L_1 et L_2 et on donne une expression régulière pour $L_1 \cup L_2$, $L_1.L_2$ et L_1^* .

$r1$ et $r1$ sont réduites.

L_1	L_2	$L_1 \cup L_2$	$L_1.L_2$	L_1^*
0	0	0	0	1
0	1	1	0	1
0	r2	r2	0	1
0	r2+1	r2+1	0	1
1	0	1	0	1
1	1	1	1	1
1	r2	r2+1	r2	1
1	r2+1	r2+1	r2+1	1
r1	0	r1	0	r1*
r1	1	r1+1	r1	r1*
r1	r2	r1+r2	r1.r2	r1*
r1	r2+1	r1+r2+1	r1.r2+r1	r1*
r1+1	0	r1+1	0	r1*
r1+1	1	r1+1	r1+1	r1*
r1+1	r2	r1+r2+1	r1.r2+r2	r1*
r1+1	r2+1	r1+r2+1	r1.r2+r1+r2+1	r1*

Éléments de réponse pour l'exercice 12

- $(a+b)^*.a.(a+b)^*$
- $b^*.b^*.a.b^*$
- $(a.a+b)^*$
- $((a+b).(a+b).(a+b))^*. (a+b+a.a+a.b+b.a+b.b)$
- $(1+b).(a.b)^*. (1+a)$ ou $(a.b)^* + a.(a.b)^* + a.(a.b)^*.b + (a.b)^*.b$
- S'il y a un a avant un b alors il y a un facteur ab , sinon il y a un facteur ba d'où l'expression régulière $(a+b)^*. (a.b+b.a).(a+b)^*$
 On peut aussi distinguer la première lettre : $(a.a^*.b+b.b^*.a).(a+b)^*$

Éléments de réponse pour l'exercice 13

On utilise un résultat ci-dessus. On isole les lettres c : les mots de L sont de la forme $u_0.c.u_1.c.\dots.u_{n-1}.c.u_n$ où u_i est un mot du langage sur l'alphabet $\{a, b\}$ tels que deux lettres consécutives sont toujours distinctes.
 On note $r1=(1+b).(a.b)^*. (1+a)$.
 On doit avoir $u_i \neq \epsilon$ pour $1 \leq i \leq n - 1$ dans ce cas l'ensemble peut être dénoté par $r2 = a.(b.a)^*(1+b)+b.(a.b)^*. (1+a)$
 Une expression régulière dénotant le langage est donc $r1.(c.r2)^*.c.r1+r1$.

Éléments de réponse pour l'exercice 14

Le complémentaire est l'ensemble des mots qui ne contiennent pas aba .

Si on isole les a et les b toute suite non vide de a doit être suivie d'une suite d'au moins deux b donc les facteurs de base sont de la forme $a^n b^p$ avec $n \geq 1$ et $p \geq 2$. De plus on peut commencer par des b et finir par des a puis des b d'où \bar{L} est dénoté par $b^* \cdot (a \cdot a^* \cdot b \cdot b \cdot b^*)^* \cdot a^* \cdot b^*$.

Éléments de réponse pour l'exercice 15

- Pour la seconde expression on isole les lettres b ; elles sont séparées par des a^k .
Pour la troisième on a séparé les mots ne contenant pas de b .
Pour la dernière on isole les lettres.
- Les mots du langage dénoté par $(r1 \cdot r2)^*$ sont soit le mot vide soit de la forme $u_1 \cdot v_1 \cdot u_2 \cdot v_2 \cdots u_n \cdot v_n$ avec $u_1 \in L[r1]$, $v_n \in L[r2]$ et $v_1 \cdot u_2 \cdot v_2 \cdots u_n \in L[(r1 \cdot r2)^*]$.

Éléments de réponse pour l'exercice 16

- $P = \{a, b\}$, $F = \{aa\}$, $S = \{a\}$
- $P = \{a\}$, $F = \{aa\}$, $S = \{a, b\}$
- $P = \{a\}$, $F = \{aa, ab\}$, $S = \{a\}$
- $P = \{a\}$, $F = \{aa, ba\}$, $S = \{a\}$
- $P = \{a\}$, $F = \{aa, bb\}$, $S = \{a\}$
- ...

Éléments de réponse pour l'exercice 17

On note $L_1 = (P \cdot \mathcal{A}^* \cup \mathcal{A}^* \cdot S) \setminus \mathcal{A}^* \cdot N \cdot \mathcal{A}^*$ et L_2 le langage local défini par $P, S, F = \mathcal{A}^2 \setminus N$ et $\varepsilon \notin L$.
On va prouver que $L_1 = L_2$.

- Si $u = u_1 u_2 \cdots u_n \in L_1$ alors $u = u_1 v \in P \cdot \mathcal{A}^*$ donc $u_1 \in P$ et $u = w u_n \in \mathcal{A}^* \cdot S$ donc $u_n \in S$.
Si on n'avait pas $u_i u_{i+1} \in F$ alors on aurait $u_i u_{i+1} \in N$ donc $u = u' u_i u_{i+1} u'' \in \mathcal{A}^* \cdot N \cdot \mathcal{A}^*$ ce qui est impossible. Ainsi $u_i u_{i+1} \in F$ pour tout i d'où $u \in L_2$. On a prouvé $L_1 \subset L_2$.
- Si $u = u_1 u_2 \cdots u_n \in L_2$ alors $u_1 \in P$ donc $u \in P \cdot \mathcal{A}^*$ et $u_n \in S$ donc $u \in \mathcal{A}^* \cdot S$.
Si on avait $u \in \mathcal{A}^* \cdot N \cdot \mathcal{A}^*$ alors il existerait i tel que $u_i u_{i+1} \in N = \mathcal{A}^2 \setminus F$ ce qui est impossible.
Ainsi $u \in (P \cdot \mathcal{A}^* \cup \mathcal{A}^* \cdot S) \setminus \mathcal{A}^* \cdot N \cdot \mathcal{A}^* = L_1$. On a prouvé $L_2 \subset L_1$.

Éléments de réponse pour l'exercice 18

Si $L_1 \setminus \{\varepsilon\}$ et $L_2 \setminus \{\varepsilon\}$ sont associés aux triplets (P_1, S_1, F_1) et (P_2, S_2, F_2) alors $(L_1 \cap L_2) \setminus \{\varepsilon\}$ est local avec les paramètres $(P_1 \cap P_2, S_1 \cap S_2, F_1 \cap F_2)$.

Éléments de réponse pour l'exercice 19

Si $L \setminus \{\varepsilon\}$ est défini par (P, S, F) alors L^* contient ε et est défini par $(P, S, F \cup S \cdot F)$.

Éléments de réponse pour l'exercice 20

L_1 est défini par $(\{a\}, \{b\}, \{ab\})$ et L_2 par $(\{a\}, \{a\}, \{aa\})$. Aucun ne contient ε .

\hookrightarrow Si $L_1 \cdot L_2 = \{aba^n ; n \in \mathbb{N}\}$ était local on aurait $P = \{a\}$, $S = \{b, a\}$ et $F = \{ab, ba, aa\}$ donc $L_1 \cdot L_2$ devrait contenir $abab$ ce qui n'est pas le cas : il n'est pas local.

↪ Si $L_1 \cup L_2$ était local alors on aurait $P = \{a\}$, $S = \{b, a\}$ et $F = \{ab, ba, aa\}$. Ainsi aab appartiendrait à $L_1 \cup L_2$ ce qui n'est pas le cas : $L_1 \cup L_2$ n'est pas local.

Éléments de réponse pour l'exercice 21

L_1 est défini par (S_1, P_1, F_1) et L_2 est défini par (S_2, P_2, F_2) .

$L_1 \cup L_2$ est inclus dans le langage défini par $(S_1 \cup S_2, P_1 \cup P_2, F_1 \cup F_2)$.

Soit $u = x_1 x_2 \dots x_n$ appartenant au langage local défini par $(S_1 \cup S_2, P_1 \cup P_2, F_1 \cup F_2)$.

Si $x_1 \in \mathcal{A}_1$ alors $x_1 x_2 \in F_1 \cup F_2$ impose $x_1 x_2 \in F_1$ donc $x_2 \in \mathcal{A}_1$ et toutes les autres lettres appartiennent à \mathcal{A}_1 donc $u \in L_1$.

De même si $x_1 \in \mathcal{A}_2$ alors $u \in L_2$. Ainsi $u \in L_1 \cup L_2$ d'où l'égalité demandée.

↪ Si $\epsilon \notin L_1$ et $\epsilon \notin L_2$ alors $L_1.L_2$ est défini par $(P_1, S_2, F_1 \cup F_2 \cup S_1.P_2)$.

↪ Si $\epsilon \in L_1$ et $\epsilon \notin L_2$ alors $L_1.L_2$ est défini par $(P_1 \cup P_2, S_2, F_1 \cup F_2 \cup S_1.P_2)$.

↪ Si $\epsilon \notin L_1$ et $\epsilon \in L_2$ alors $L_1.L_2$ est défini par $(P_1, S_1 \cup S_2, F_1 \cup F_2 \cup S_1.P_2)$.

↪ Si $\epsilon \in L_1$ et $\epsilon \in L_2$ alors $L_1.L_2$ est défini par $(P_1 \cup P_2, S_1 \cup S_2, F_1 \cup F_2 \cup S_1.P_2)$.

Éléments de réponse pour l'exercice 22

On procède par induction structurelle sur l'expression régulière.

↪ Si r est 0, 1 ou a avec $a \in \mathcal{A}$ alors $L[r]$ est local.

↪ Si $r = (r1^*)$ et si r est linéaire alors $r1$ est linéaire.

Si $L[r1]$ est local on a vu qu'alors $L[r] = (L[r1])^*$ est local.

↪ Si $r = (r1+r2)$ et si r est linéaire alors $r1$ et $r2$ sont linéaires.

Si $L[r1]$ et $L[r2]$ sont locaux alors, comme les lettres de $r1$ et de $r2$ appartiennent à des ensembles distincts, l'exercice 21 montre que $L[r] = L[r1] \cup L[r2]$ est local.

↪ De même si $r = (r1.r2)$

Éléments de réponse pour l'exercice 23

```
let rec lvide r =
  match r with
  | Zero -> true
  | Un -> false
  | Lettre _ -> false
  | Somme(e1,e2) -> lvide e1 && lvide e2
  | Produit(e1,e2) -> lvide e1 || lvide e2
  | Etoile e -> false;;
```

Éléments de réponse pour l'exercice 24

```
let rec motVide r =
  match r with
  |Zero -> false
  |Un -> true
  |Lettre _ -> false
  |Somme(r1, r2) -> motVide r1 || motVide r2
  |Produit(r1, r2) -> motVide r1 && motVide r2
  |Etoile r -> true;;
```

Éléments de réponse pour l'exercice 25

```
let rec prefixes r =
  match r with
  |Zero -> []
  |Un -> []
  |Lettre a -> [a]
  |Somme(r1, r2) -> (prefixes r1)@(prefixes r2)
  |Produit(r1, r2) -> if motVide r1
                       then (prefixes r1)@(prefixes r2)
                       else prefixes r1
  |Etoile r -> prefixes r;;
```

```
let rec suffixes r =
  match r with
  |Zero -> []
  |Un -> []
  |Lettre a -> [a]
  |Somme(r1, r2) -> (suffixes r1)@(suffixes r2)
  |Produit(r1, r2) -> if motVide r2
                       then (suffixes r1)@(suffixes r2)
                       else suffixes r1
  |Etoile r -> suffixes r;;
```

```
let rec assembler liste1 liste2 =  
  let coller x y = (string_of_char x)^(string_of_char y) in  
  let ajouter x = map (coller x) liste2 in  
  match liste1 with  
  |[] -> []  
  |t::q -> (ajouter t)@(assembler q liste2);;
```

```
let rec facteurs r =  
  match e with  
  |Zero -> []  
  |Un -> []  
  |Lettre a -> []  
  |Somme(r1, r2) -> (facteurs r1)@(facteurs r2)  
  |Produit(r1, r2) -> ((facteurs r1)@(facteurs r2))  
    @(assembler (suffixes r1) (prefixes r2))  
  |Etoile r -> (facteurs r)@(assembler (suffixes r) (prefixes r));;
```

8 Automates

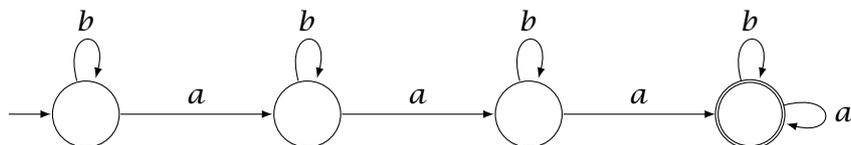
Éléments de réponse pour l'exercice 1

La propriété se démontre par récurrence sur $|v|$.

1. Elle est triviale pour $|v| = 0$ car alors $v = \epsilon$ et $s.(u.\epsilon) = s.u = (s.u).\epsilon$.
2. On suppose qu'elle est vraie pour les mots de longueur n .
 Si v est de longueur $n + 1$ on écrit $v = v'.x$ avec v' de longueur n .
 On a $s.(u.v') = (s.u).v'$ d'après l'hypothèse de récurrence. Alors, en notant $s' = s.u$,
 $s.(u.v) = s.(u.v'.x) = (s.(u.v')).x = ((s.u).v').x = (s'.v').x = s'.(v'.x) = s'.v = (s.u).v$.
3. la propriété est donc vraie pour tout n .

Éléments de réponse pour l'exercice 2

L est dénoté par $(b^* . a) . (b^* . a) . (b^* . a) . (a+b)^*$ est reconnu par



Éléments de réponse pour l'exercice 3

Si on veut calculer le reste de la division par 3 d'un nombre écrit en binaire, à chaque nouveau bit lu le nombre est multiplié par 2 et on ajoute le bit :

Reste	0	1	2
0	0	2	1
1	1	0	2

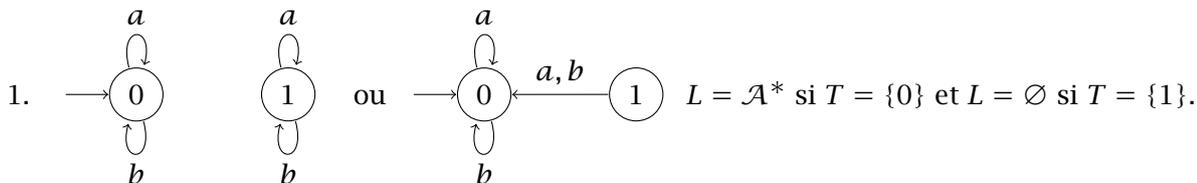
$0.n = i$ est donc équivalent à $n \bmod 3 = i$.

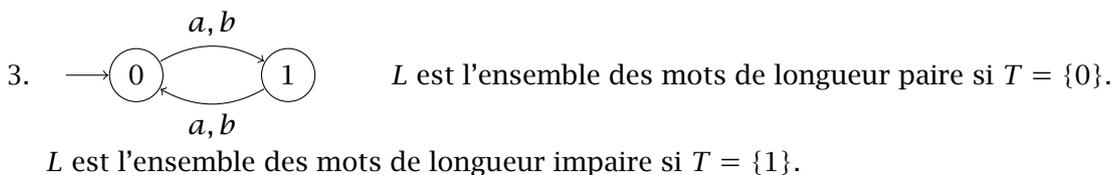
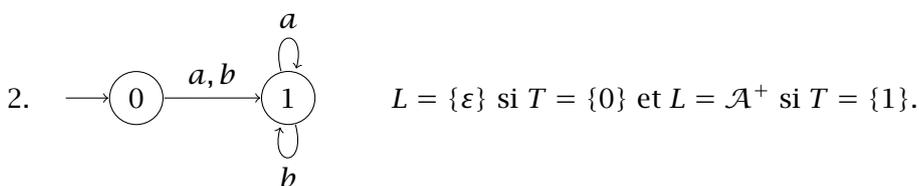
Éléments de réponse pour l'exercice 4

On note $S = \{0, 1\}$, il y a deux choix pour chaque transition $0.a, 0.b, 1.a$ et $1.b$ donc il y a 16 machines possibles.

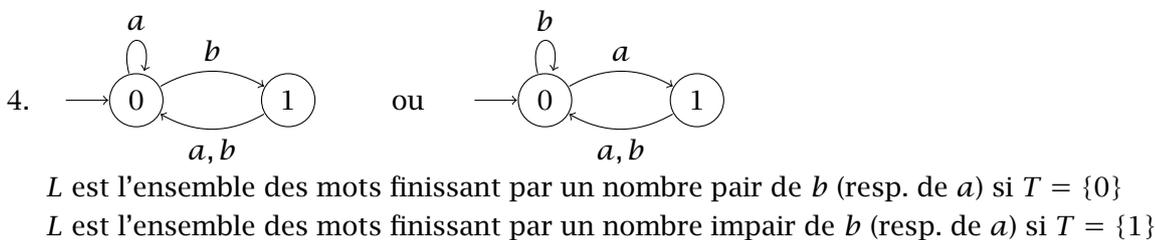
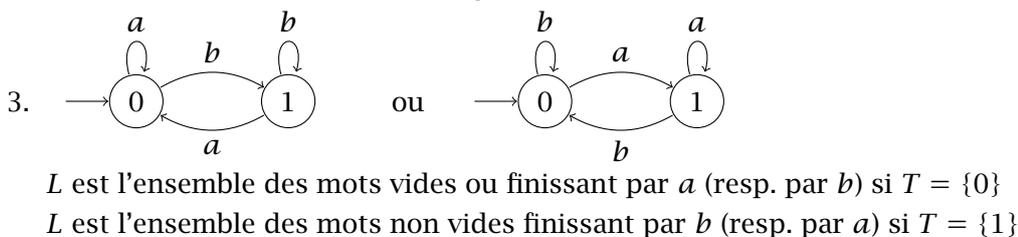
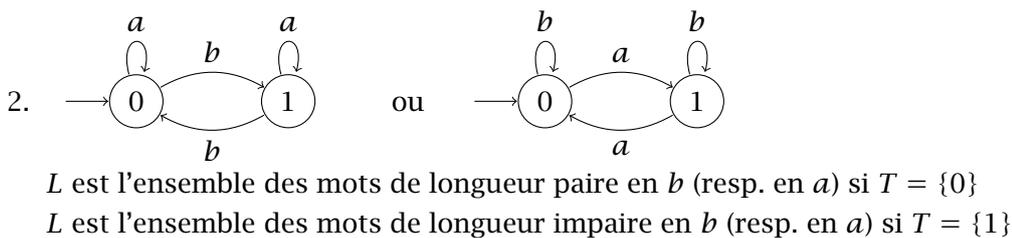
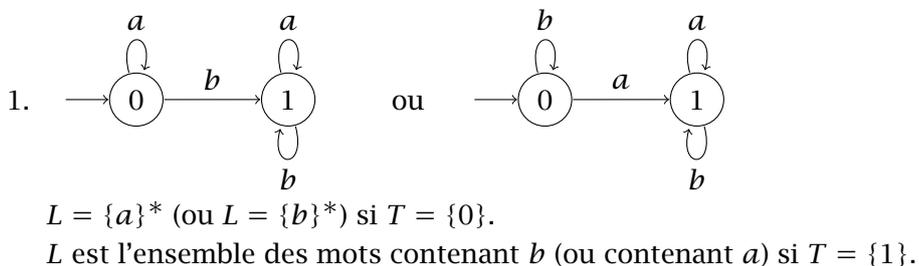
On choisira 0 pour état initial ; il reste à choisir T . $T = \emptyset$ ou $T = \{0, 1\}$ donnent des langages triviaux (\emptyset ou \mathcal{A}^*), on aura deux langages complémentaires en choisissant $T = \{0\}$ ou $T = \{1\}$.

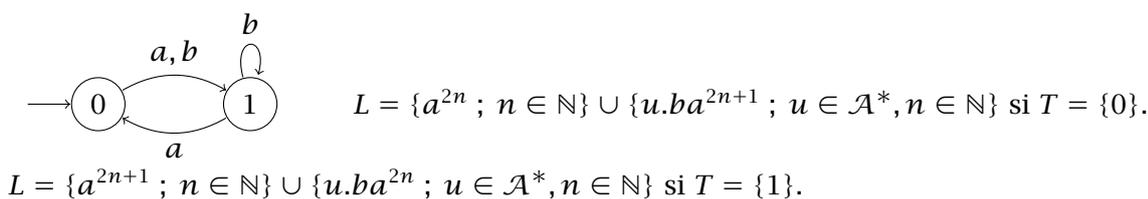
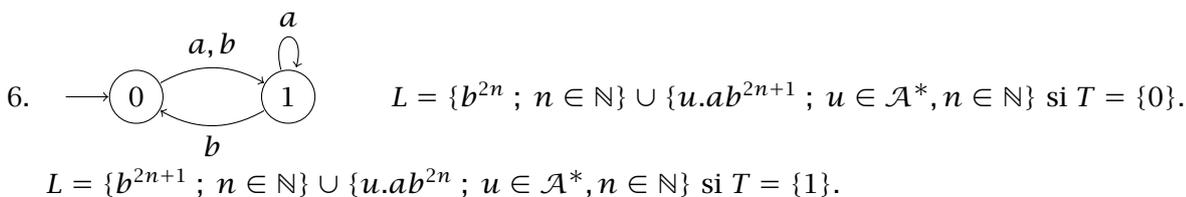
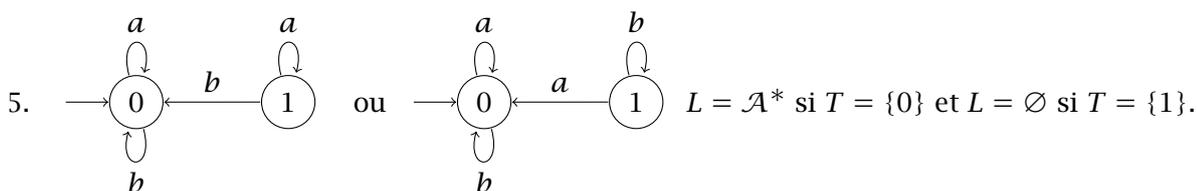
On commence par les 4 machines symétriques en a et b .





Dans les autres cas on peut regrouper les automates par paires en échangeant le rôle de a et de b .





Éléments de réponse pour l'exercice 5

On part de l'état initial, noté 1, et on numérote les états tant que l'image par δ est un nouvel état. Le dernier état accessible est envoyé dans un état déjà visité.

$$1 \xrightarrow{a} 2 \xrightarrow{a} \dots \xrightarrow{a} n \xrightarrow{a} p \text{ avec } 1 \leq p \leq n$$

On obtient une forme de poêle.

Si les états finaux sont $f_1 < f_2 < \dots < f_r$ avec $f_s < p \leq f_{s+1}$ alors le langage reconnu est $\{a^n ; n \in \mathbb{N}\}$ avec, en notant $T = n - p + 1$, $A = \{f_1, f_2, \dots, f_s\} \cup \{f_{s+1} + Tn, f_{s+2} + Tn, \dots, f_r + Tn ; n \in \mathbb{N}\}$.

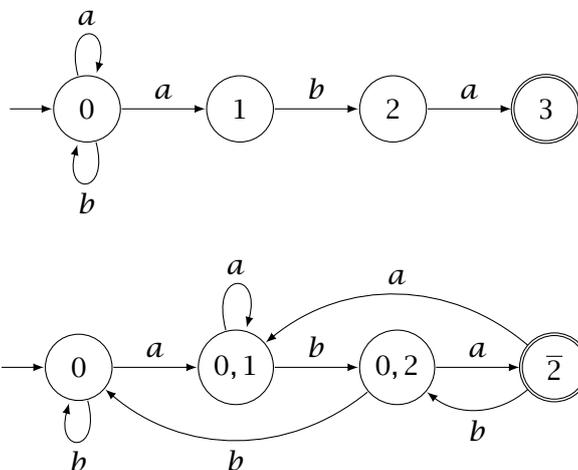
Éléments de réponse pour l'exercice 6

1. Si tous les états sont co-accessibles alors, pour tout u , $s_0.u$ est co-accessible donc il existe v tel que $(s_0.u).v \in T$ donc $u.v$ est un mot reconnu. On a prouvé que tout mot est un préfixe d'un mot reconnu.
2. On suppose que tout mot est un préfixe d'un mot reconnu.
 Pour tout état accessible s , on peut écrire $s = s_0.u$. u est un préfixe d'un mot reconnu u' , $u' = u.v$. On a $s_0.u' \in T$ donc $s.v = (s_0.u).v \in T$.
 Tout état accessible est donc co-accessible.

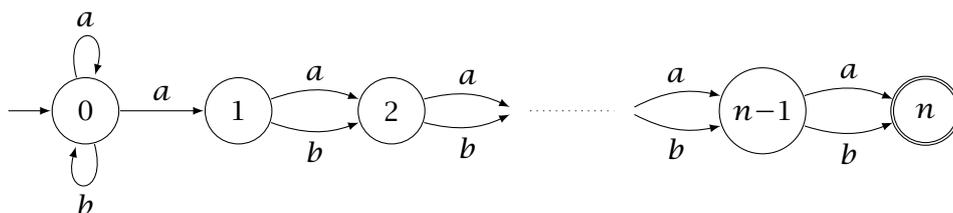
Éléments de réponse pour l'exercice 7

1. Si $\varepsilon \in L(Q)$ alors il existe un état s de S qui est initial et final à la fois. Ainsi $I \cap T \neq \emptyset$. Dans Q' on a donc $I \in \mathcal{T}$ donc $\varepsilon \in L(Q')$.
2. Inversement; si $\varepsilon \in L(Q')$, alors l'état initial, I , est final donc il existe un état s dans $I \cap T$. s est initial et final dans M donc $\varepsilon \in L(Q)$.

Éléments de réponse pour l'exercice 8



Éléments de réponse pour l'exercice 9



Soit $Q = (\mathcal{A}, S, \delta, s_0, T)$ un automate déterministe qui reconnaît \mathcal{A} .

On définit φ de \mathcal{A}^n vers S par $\varphi(u) = s_0.u$.

On montre que φ est injective. On suppose qu'on a $\varphi(u) = \varphi(u')$.

Si on a $u \neq u'$ on considère la première position k telle que $u_k \neq u'_k$: on a $1 \leq k \leq n$, $u = v.a.w$, $u' = v.b.w'$ (ou l'inverse) avec $|v| = k - 1$ et $|w| = |w'| = n - k$.

On a $u.a^{k-1} = v.a.(w.a^{k-1})$ qui appartient à L car $|w.a^{k-1}| = n - 1$ mais $u'.a^{k-1}$ n'appartient pas à L .

Ainsi $\varphi(u).a^{k-1} = s_0.(u.a^{k-1}) \in T$ mais $\varphi(u').a^{k-1} = s_0.(u'.a^{k-1}) \notin T$ ce qui rend impossible l'égalité $\varphi(u) = \varphi(u')$.

On doit donc avoir $u = u'$: φ est injective.

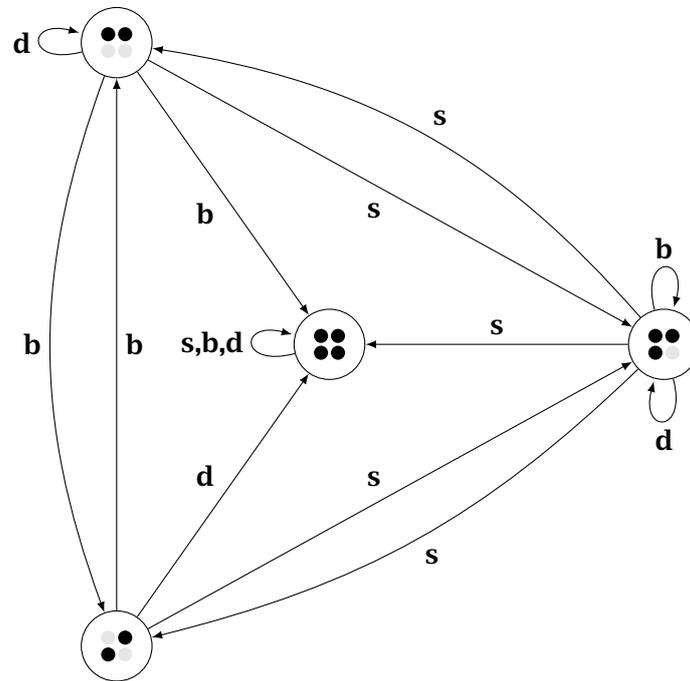
On a une injection de \mathcal{A}^n dans S donc $|S| \geq |\mathcal{A}^n| = 2^n$.

Éléments de réponse pour l'exercice 10

Les calculs dans Q^T sont les inverses des calculs dans Q .

Éléments de réponse pour l'exercice 11

Voici un automate décrivant les mouvements :



- ↪ **s** (pour singleton) consiste à retourner 1 jeton (ou 3 jetons)
- ↪ **b** (pour bord) consiste à retourner 2 jeton adjacents
- ↪ **d** (pour diagonale) consiste à retourner 2 jeton en diagonale.
- ↪ Depuis l'état de 4 couleurs identiques les transformations renvoie au même état car on est dans un état gagnant.

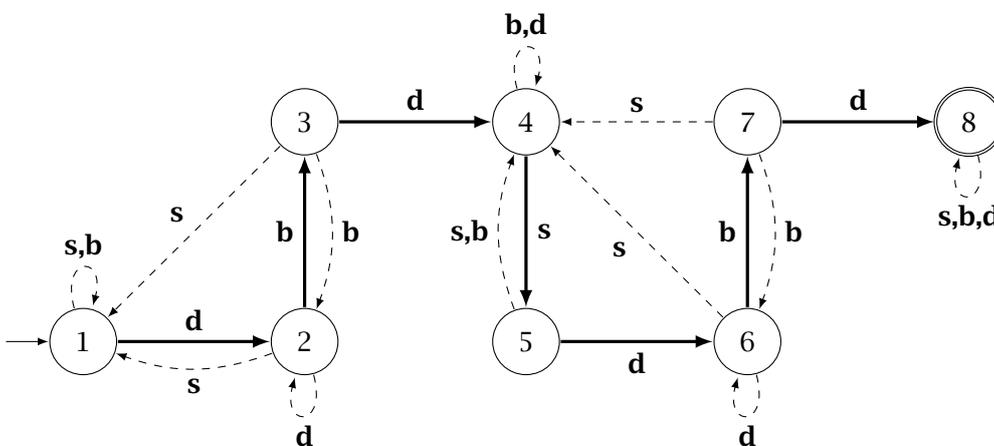
On note

- ↪ *T* l'état dans lequel tous les jetons sont de la même couleur,
- ↪ *U* l'état dans lequel un jeton a une couleur et les 3 autres ont l'autre couleur,
- ↪ *B* l'état dans lequel 2 jetons côte-à-côte ont une couleur et les 2 autres ont l'autre couleur,
- ↪ *D* l'état dans lequel 2 jetons en diagonale ont une couleur et les 2 autres ont l'autre couleur.

La détermination, avec *T*, *U*, *P* et *D* états initiaux, donne

	<i>U, B, D, T</i>	<i>U, B, T</i>	<i>U, D, T</i>	<i>U, T</i>	<i>B, D, T</i>	<i>B, T</i>	<i>D, T</i>	<i>T</i>
s	<i>U, B, D, T</i>	<i>U, B, D, T</i>	<i>U, B, D, T</i>	<i>B, D, T</i>	<i>U, T</i>	<i>U, T</i>	<i>U, T</i>	<i>T</i>
b	<i>U, B, D, T</i>	<i>U, D, T</i>	<i>U, B, T</i>	<i>U, T</i>	<i>U, T</i>	<i>D, T</i>	<i>B, T</i>	<i>T</i>
d	<i>U, B, T</i>	<i>U, B, T</i>	<i>U, T</i>	<i>U, T</i>	<i>B, T</i>	<i>B, T</i>	<i>T</i>	<i>T</i>
état	1	2	3	4	5	6	7	8

L'état final qui nous intéresse est $\{T\}$: on veut être certain d'arriver.



La suite de mouvement **dbdsdbd** permet d'arriver au résultat.

Éléments de réponse pour l'exercice 12

Les mots non vides de L , $u = x_1x_2 \dots x_n$, sont associés aux mots reconnus par Q selon le chemin $\varepsilon \xrightarrow{x_1} x_1 \xrightarrow{x_2} x_2 \xrightarrow{x_3} \dots \xrightarrow{x_{n-1}} x_{n-1} \xrightarrow{x_n} x_n$.

Inversement un tel chemin a une étiquette u dans L car la transition $\varepsilon \xrightarrow{x_1} x_1$ implique $x_1 \in P$, les $x_{i-1} \xrightarrow{x_i} x_i \xrightarrow{x_{i+1}} x_{i+1}$ impliquent $x_i x_{i+1} \in F$ et x_n terminal implique $x_n \in T$.

Le mot vide est reconnu si et seulement si ε est un état final.

Éléments de réponse pour l'exercice 13

$Q = (\mathcal{A}, S, \delta, s_0, T)$ est un automate local. On définit

- $\hookrightarrow P = \{x \in \mathcal{A} ; \delta(s_0, x) \text{ est défini}\},$
- $\hookrightarrow S = \{x \in \mathcal{A} ; \exists s \in S, \delta(s, x) \in T\},$
- $\hookrightarrow T = \{xy \in \mathcal{A}^2 ; \exists s, t \in S, \delta(s, x) = t \text{ et } \delta(t, y) \text{ est défini}\}.$

On note L le langage local défini par (P, S, F) qui contient ou non ε selon que s_0 est final ou non.

1. Par construction ε appartient à L et $L(Q)$ ou a aucun des deux.
2. Si un mot non vide est reconnu par Q alors sa première lettre est dans P , sa dernière lettre est dans S et ses facteurs (s'il y en a) sont dans F , par construction.
Ainsi $L(Q) \setminus \{\varepsilon\}$ est inclus dans le, noté L .
3. Inversement supposons que u appartienne au langage local défini par (P, S, F) avec $u \neq \varepsilon$.
On écrit $u = x_1x_2 \dots x_n$.
 $x_1 \in P$ donc $\delta(s_0, x_1)$ est défini : on note $s_1 = \delta(s_0, x_1)$.
 $x_1x_2 \in F$ donc il existe une transition $s \xrightarrow{x_1} t \xrightarrow{x_2} r$. On a donc $t = \delta(s, x_1)$ et $r = \delta(t, x_2)$.
Comme $\delta(s, x_1)$ et $\delta(s_0, x_1)$ sont définis dans l'automate local, ils sont égaux donc $t = s_1$ et on peut définir $s_2 = r = \delta(s_1, x_2)$.

On peut ainsi définir un chemin dans Q d'étiquette u qui aboutit dans T car on a $x_n \in S$.
Ainsi u est reconnu par $Q : L \setminus \{\varepsilon\} \subset L(Q)$.

On a donc $L = L(Q)$.

Éléments de réponse pour l'exercice 14

1. On montre par récurrence sur n que, pour tout mot $u \in \mathcal{B}^*$ et pour tout mot $v \in \mathcal{B}^*$ de longueur n , $f^*(u.v) = f^*(u).f^*(v)$.
2. On a $L_1 \subset L_1 \cup L_2$ donc $f^*(L_1) \subset f^*(L_1 L_1 \cup L_2)$.
De même $f^*(L_1) \subset f^*(L_1 L_1 \cup L_2)$ donc $f^*(L_1) \cup f^*(L_2) \subset f^*(L_1 \cup L_2)$.
Inversement si $v \in f^*(L_1 \cup L_2)$ alors il existe $u \in L_1 \cup L_2$ tel que $v = f^*(u)$. Si $u \in L_i$ alors $v \in f^*(L_i) \subset f^*(L_1) \cup f^*(L_2)$ d'où l'égalité.
3. $f^*(L_1.L_2) = \{f^*(u.v) ; u \in L_1, v \in L_2\} = \{f^*(u).f^*(v) ; u \in L_1, v \in L_2\}$ puis
 $f^*(L_1.L_2) = \{f^*(u) ; u \in L_1\} \cdot \{f^*(v) ; v \in L_2\} = f^*(L_1).f^*(L_2)$.
4. On en déduit par récurrence sur n que $f^*(L_1^n) = (f^*(L_1))^n$ puis
 $f(L_1^*) = f\left(\bigcup_{n \in \mathbb{N}} L_1^n\right) = \bigcup_{n \in \mathbb{N}} f(L_1^n) = \bigcup_{n \in \mathbb{N}} (f^*(L_1))^n = (f^*(L_1))^*$.
5. On a $L_1 \cap L_2 \subset L_1$ donne $f^*(L_1 \cap L_2) \subset f^*(L_1)$ et, de même, $f^*(L_1 \cap L_2) \subset f^*(L_2)$ donc
 $f^*(L_1 \cap L_2) \subset f^*(L_1) \cap f^*(L_2)$.
Cependant l'inclusion peut être stricte. Pour $\mathcal{B} = \mathcal{A} = \{a, b\}$, $f(a) = f(b) = a$, $L_1 = \{a\}^*$ et
 $L_2 = \{b\}^*$ on a $L_1 \cap L_2 = \{\varepsilon\}$ mais $f^*(L_1) = f^*(L_2) = \{a\}^*$ donc
 $f^*(L_1 \cap L_2) = \{\varepsilon\} \neq \{a\}^* = f^*(L_1) \cap f^*(L_2)$.

Éléments de réponse pour l'exercice 15

La démonstration se fait par induction structurelle.

1. Les cas de base sont immédiats.
2. Pour $r = x$, $L[\hat{\varphi}(x)] = L[\varphi(x)] = \{\varphi(x)\} = \varphi^*(\{x\}) = \varphi^*(L[x])$.
3. On suppose que $L[\hat{\varphi}(e)] = \varphi^*(L[e])$ et $L[\hat{\varphi}(f)] = \varphi^*(L[f])$.
 $L[\hat{\varphi}(e+f)] = L[\hat{\varphi}(e) + \hat{\varphi}(f)] = L[\hat{\varphi}(e)] \cup L[\hat{\varphi}(f)] = \varphi^*(L[e]) \cup \varphi^*(L[f]) = \varphi^*(L[e] \cup L[f]) = \varphi^*(L[(e+f)])$
De même pour le produit et l'étoile.

Éléments de réponse pour l'exercice 16

Tout mot de $L(Q)$ est l'étiquette d'un calcul réussi dans Q ; on note $u = x_1 x_2 \dots x_n$,
 $s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} s_2 \xrightarrow{x_3} \dots \xrightarrow{x_{n-1}} s_{n-1} \xrightarrow{x_n} x_n$.

Dans $\hat{f}(Q)$ le chemin devient

$$s_0 \xrightarrow{f(x_1)} s_1 \xrightarrow{f(x_2)} s_2 \xrightarrow{f(x_3)} \dots \xrightarrow{f(x_{n-1})} s_{n-1} \xrightarrow{f(x_n)} x_n$$

il reste réussi car les ensembles initiaux et terminaux sont inchangés ;

ainsi $f(x_1)f(x_2)\dots f(x_n) = f^*(u)$ appartient à $L(\hat{f}(Q))$ d'où $f^*(L(Q)) \subset L(\hat{f}(Q))$.

De même tout calcul réussi dans $\hat{f}(Q)$ peut se "remonter" en un calcul réussi dans Q car les étiquettes sont des $f(x_i)$. Le mot associé est donc l'image par f^* d'un mot de $L(Q)$ d'où $L(\hat{f}(Q)) \subset f^*(L(Q))$

On a ainsi $L(\hat{f}(Q)) = f^*(L(Q))$.

9 Compléments : automates

Éléments de réponse pour l'exercice 1

On a une matrice de $N \times N$ termes qu'on doit calculer N fois. Chaque opération est en temps constant donc la complexité est un $\mathcal{O}(N^3)$.

Éléments de réponse pour l'exercice 2

1. v appartient à $u^{-1}.L$ si et seulement si $u.v$ appartient à L donc $\varepsilon \in u^{-1}.L$ si et seulement si $u = u.\varepsilon \in L$.
2. $u \in \varepsilon^{-1}.L$ si et seulement si $u = \varepsilon.u \in L$; $\varepsilon^{-1}.L = L$.
3. $u^{-1}.\mathcal{A}^* \subset \mathcal{A}^*$; si $v \in \mathcal{A}^*$ alors $u.v \in \mathcal{A}^*$ donc $v \in u^{-1}.\mathcal{A}^*$ d'où $\mathcal{A}^* \subset u^{-1}.\mathcal{A}^*$ puis $\mathcal{A}^* = u^{-1}.\mathcal{A}^*$.
4. $w \in v^{-1}.(u^{-1}.L) \iff v.w \in u^{-1}.L \iff u.(v.w) \in L \iff (u.v).w \in L \iff w \in (u.v)^{-1}.L$.

Éléments de réponse pour l'exercice 3

En comptant le nombre de b on montre que $a^{-1}.L_1 = L_1$, $b^{-1}.L_1 = L_2$, $a^{-1}.L_2 = L_2$ et $b^{-1}.L_2 = L_1$.

En appliquant le résultat ci-dessus on en déduit que

$u^{-1}.L_1 = L_1$ et $u^{-1}.L_2 = L_2$ si $u \in L_1$; $u^{-1}.L_1 = L_2$ et $u^{-1}.L_2 = L_1$ si $u \in L_2$.

Éléments de réponse pour l'exercice 4

$u \in (a^p)^{-1}.L \iff a^p.u \in L \iff \exists n \in N, a^p.u = a^n.b^n \iff u = a^{n-p}.b^n$

$(a^p.b^q)^{-1}.L = (b^q)^{-1}.\left((a^p)^{-1}.L\right)$ or le seul mot de $(a^p)^{-1}.L$ qui commence par un b est b^p . L'ensemble est non vide si et seulement si $p \geq q$ et alors son seul élément est b^{p-q} .

Éléments de réponse pour l'exercice 5

1. Si u appartient à $a^{-1}(L_1 \cup L_2)$ alors $a.u$ appartient à $L_1 \cup L_2$ donc

soit $a.u \in L_1$ d'où $u \in a^{-1}L_1 \subset a^{-1}L_1 \cup a^{-1}L_2$,

soit $a.u \in L_2$ d'où $u \in a^{-1}L_2 \subset a^{-1}L_1 \cup a^{-1}L_2$.

Dans les deux cas on a $u \in a^{-1}L_1 \cup a^{-1}L_2$ d'où $a^{-1}(L_1 \cup L_2) \subset a^{-1}L_1 \cup a^{-1}L_2$.

Inversement on a $L_1 \subset L_1 \cup L_2$ donc $a^{-1}L_1 \subset a^{-1}(L_1 \cup L_2)$.

De même $a^{-1}L_2 \subset a^{-1}(L_1 \cup L_2)$ d'où $a^{-1}L_1 \cup a^{-1}L_2 \subset a^{-1}(L_1 \cup L_2)$.

On en déduit l'égalité demandée.

2. Soit u appartenant à $(a^{-1}.L_1).L_2$.

$u = u_1.u_2$ avec $u_2 \in L_2$ et $u_1 \in a^{-1}.L_1$ donc $a.u_1 \in L_1$.

On a alors $a.u = (a.u_1).u_2 \in L_1.L_2$ donc $u \in a^{-1}.(L_1.L_2)$.

On a prouvé $(a^{-1}.L_1).L_2 \subset a^{-1}.(L_1.L_2)$ sans condition sur L_1 .

Inversement si u appartient à $a^{-1}.(L_1.L_2)$ alors $a.u \in L_1.L_2$

donc on peut écrire $a.u = u_1.u_2$ avec $u_1 \in L_1$ et $u_2 \in L_2$.

Comme u_1 ne peut être le mot vide, il doit commencer par a donc on peut écrire $u_1 = a.u'_1$.
On a alors $u'_1 \in a^{-1}.L_1$ puis $u = u'_1.u_2 \in (a^{-1}.L_1).L_2$ ce qui prouve la seconde inclusion puis l'égalité.

3. Prouver que si $\varepsilon \in L_1$ alors $L_2 \subset L_1.L_2$ donc $a^{-1}L_2 \subset a^{-1}.(L_1.L_2)$.

L'inclusion $(a^{-1}.L_1).L_2 \subset a^{-1}.(L_1.L_2)$ prouvée ci-dessus donne alors
 $a^{-1}.L_2 \cup (a^{-1}.L_1).L_2 \subset a^{-1}.(L_1.L_2)$

Comme ci-dessus on écrit, pour $u \in a^{-1}.(L_1.L_2)$ $a.u = u_1.u_2$.

Si $u_1 \neq \varepsilon$ on a encore $u \in (a^{-1}.L_1).L_2$.

Si $u_1 = \varepsilon$ on a $a.u = u_2 \in L_2$ donc $u \in a^{-1}L_2$.

On a donc $u \in a^{-1}.L_2 \cup (a^{-1}.L_1).L_2$ ce qui prouve la seconde inclusion puis l'égalité.

4. Prouver que $a^{-1}.(L_1^*) = (a^{-1}.L_1).L_1^*$.

Éléments de réponse pour l'exercice 6

On va prouver que π est un isomorphisme d'automates.

1. On a vu que π est surjective de Λ vers S .

Comme ces deux ensembles sont finis de même cardinal, π est une bijection.

Si $\lambda = u^{-1}.L$ on a $\lambda = \pi(s_0.u)$ donc $\pi^{-1}(\lambda) = s_0.u$: le résultat est indépendant de u .

On a prouvé, dans ce cas, la réciproque de l'exercice précédent

$u^{-1}.L = v^{-1}.L$ implique $s_0.u = s_0.v$.

2. Si s est terminal alors $s = s_0.u$ avec $u \in L$.

On a donc $\pi(s) = u^{-1}.L$ qui contient ε : $\pi(s)$ appartient à Λ_T .

Inversement si $\pi(s)$ est terminal dans Q_L alors il contient ε ; pour $s = s_0.u$ cela signifie qu'on a $\varepsilon \in u^{-1}.L$ donc $u \in L$ et $s \in T$.

On a bien $\pi(T) = \Lambda_T$.

3. Soit $s = s_0.u \in S$, $\pi(s) = u^{-1}.L$.

$\delta_L(\pi(s), x) = x^{-1}.\pi(s) = x^{-1}.(u^{-1}.L) = (u.x)^{-1}.L$

$\delta(s, x) = s.x = (s_0.u).x = s_0.(u.x)$ donc $\pi(\delta(s, x)) = \pi(s_0.(u.x)) = (u.x)^{-1}.L$.

On a bien $\delta_L(\pi(s), x) = \pi(\delta(s, x))$.

Les automates sont bien équivalents.

Éléments de réponse pour l'exercice 7

1. \equiv est définie par une égalité des images par la fonction $s \mapsto L_s$; c'est donc, comme l'égalité, une relation d'équivalence.

2. $s \in F$ est équivalent à $\varepsilon \text{ in } L_s$. Ainsi $L_s = L_t$ implique, si $s \in F$, que ε appartient à L_t donc que t est final.

3. On suppose que $s \equiv t$.

Soit $u \in L_{s,x}$, on a $(s.x).u = s.(x.u) \in F$ donc $x.u \in L_s = L_t$. On en déduit que $t.(x.u) = (t.x).u \in F$ d'où $u \in L_{t,x}$. On a prouvé $L_{s,x} \subset L_{t,x}$.

L'autre inclusion se prouve de même d'où l'égalité c'est-à-dire $s.x \equiv t.x$.

Éléments de réponse pour l'exercice 8

1. Par construction \hat{Q} un automate déterministe complet.
2. On note toujours $\sigma.u = \hat{\delta}^*(\sigma, u)$ pour tous $\sigma \in \Sigma$ et $u \in \mathcal{A}^*$.
Par récurrence sur $|u|$ on a $\sigma(s).u = \sigma(s.u)$.
3. Pour toute classe $\sigma(s)$ on a $s = s_0.u$ donc $\sigma(s) = \sigma(s_0).u = \sigma_0.u$: tout état est accessible, l'automate est émondé.
4. Si u est reconnu par \hat{Q} , on a $\sigma_0.u$ appartient à \hat{T} d'où $\sigma(s_0.u) \subset T$ puis $s_0.u \in T$: tout mot reconnu par \hat{Q} est reconnu par Q .
Inversement si u est reconnu par \hat{Q} , alors $s_0.u \in T$ donc $\sigma(s_0.u) \subset T$. On a $\sigma_0.u \in \hat{T}$, u est reconnu par \hat{Q} .
 Q et \hat{Q} reconnaissent le même langage.
5. Comme l'automate est émondé, tous les états sont accessibles donc, pour tout état $s \in S$, il existe $u \in \mathcal{A}^*$ tel que $s = s_0.u$. On a alors $L_s = u^{-1}.L$
Ainsi $s \equiv s'$ si et seulement si $L_s = L_{s'}$ dans l'ensemble des langages dérivés donc il y a une bijection entre les classes d'équivalence et les résiduels. Le cardinal est minimal.

Éléments de réponse pour l'exercice 9

1. $L^{(0)}$ est vide ou réduit à $\{\varepsilon\}$ selon que L ne contient pas ou contient ε . Or $\varepsilon \in L_s$ est équivalent à $s \in F$ d'où $s \equiv_0 t$ si et seulement si s et t sont dans F tous les deux ou aucun n'est dans F .
2. On suppose que $[s \equiv_k t \text{ et } \forall x \in \mathcal{A} \ s.x \equiv_k t.x]$.
Soit $u \in L_s$ de longueur $k+1$, $u = x.v$ avec $x \in \Sigma$: on a $s.u \in F$ or $s.u = (s.x).v$ donc $v \in L_{s,x}$.
Or $|v| = k$ donc $v \in L_{s,x}^{(k)}$, de plus $s.x \equiv_k s'.x$ d'où $v \in L_{t,x}^{(k)}$ c'est-à-dire $u = x.v \in L_t^{(k+1)}$.
Ceci prouve que $L_s^{(k+1)} \subset L_{s'}^{(k+1)}$.
L'autre inclusion se prouve de même donc on peut conclure à l'égalité : $s \equiv_{k+1} s'$.
3. On suppose que $s \equiv_{k+1} t$. On a alors immédiatement $s \equiv_k t$.
Soit $x \in \mathcal{A}$. Pour $u \in L_{s,x}^{(k)}$ on a $(s.x).u \in F$ donc $x.u \in L_s^{(k+1)} = L_t^{(k+1)}$.
On en déduit que $u \in L_{t,x}^{(k)} : L_{s,x}^{(k)} \subset L_{t,x}^{(k)}$.
L'autre inclusion se prouve de même donc $s.x \equiv_k t.x$ pour tout $x \in \mathcal{A}$.

Éléments de réponse pour l'exercice 10

1. On suppose que \equiv_{k+1} et \equiv_k sont identiques, on va montrer que \equiv_{k+2} et \equiv_{k+1} sont identiques.
La seule chose à prouver est $[s \equiv_{k+1} t] \Rightarrow [s \equiv_{k+2} t]$ car la réciproque est toujours vraie.
Si $s \equiv_{k+1} t$ alors $s.x \equiv_k t.x$ pour toute lettre x donc $s.x \equiv_{k+1} t.x$ d'après l'hypothèse de récurrence. On a alors les conditions qui impliquent $s \equiv_{k+2} t$.
2. Comme chaque passage crée au moins une classe d'équivalence supplémentaire on ne peut faire qu'au plus $n-2$ passages car on part de deux classes pour \equiv_0 et on aboutit à n classes au plus.

Liste des programmes

I.1	Recherche dans un arbre binaire de recherche	4
I.2	Insertion à une feuille dans un arbre binaire de recherche	6
I.3	Insertion à la racine dans un arbre binaire de recherche	8
I.4	Suppression de la racine dans un arbre binaire de recherche	11
I.5	Suppression d'un nœud dans un arbre binaire de recherche	11
II.1	Ajout d'un élément dans un tas	28
II.2	Recherche du fils prioritaire	29
II.3	Retrait du terme de priorité maximale dans un tas	29
IV.1	Voisins d'un sommet d'un graphe défini par une matrice d'adjacence	63
IV.2	Arêtes d'un graphe défini par une matrice d'adjacence	63
IV.3	Ajout d'une arête dans un graphe défini par un tableau d'adjacence	64
IV.4	Élimination d'une arête dans un graphe défini par un tableau d'adjacence	64
IV.5	Liste des arêtes dans un graphe défini par un tableau d'adjacence	65
IV.6	Parcours récursif	66
IV.7	Parcours en profondeur d'un graphe	69
IV.8	Calcul du chemin depuis s_0	71
IV.9	Parcours en largeur d'un graphe	72
IV.10	Calcul des composantes connexes	74
IV.11	Ordre final du parcours récursif	82
V.1	Algorithme de Floyd-Warshall	92
V.2	Algorithme de Dijkstra	94

Liste des définitions

Arbre binaire de recherche (ABR)	3
Arbres quasi-complets	23
Arbres décroissants	24
Tas	24
Complexités dans un tas	29
Proposition	38
Formules	39
Définition inductive	40
Valuation	43
Formules satisfaites	43
Formules équivalentes	45
Hérédité	45
Simplifications	46
Distributivité	46
Lois de Morgan	46
Formes normales	47
Forme normale canonique	48
Graphe	56
Graphe induit	57
Degrés	57
Graphe non orienté	58
Degré	58
Chemins	59
Cycle	59
Graphe (fortement) connexe	59
Sommets équivalents	60
Composantes (fortement) connexes	60
Plus court chemin	75
Graphe acyclique	79
Arbre	79
Arbre couvrant	80
Ordre topologique	82
Graphe transposé	84
Graphe valué	88
Poids d'un chemin	89
Existence d'un chemin de poids minimum	91
Complexité de l'algorithme de Floyd-Warshall	92
Complexité de l'algorithme de Dijkstra	95

Réseau	98
Capacité	99
Flot	99
Réseau résiduel	101
Chemin améliorant	102
Coupure	104
Théorème de Ford et Fulkerson	105
Produit	113
Monoïde	113
Simplifications	113
Produit de langages	114
Étoile	114
Ensembles des langages rationnels	115
Caractérisation	115
Induction structurelle	116
Expression régulière	118
Langage associé	118
Dénotation	119
Expressions équivalentes	119
Théorème : élimination de zéro	119
Théorème : élimination du un	120
Langage local	121
Caractérisation des langages locaux	122
Théorème : expressions linéaires	122
Lemme : union et produits sur des langages disjoints	122
Machine	130
Transition	131
Action d'un mot	131
Produit	132
Automate déterministe complet	132
Langage reconnu	133
Accessibilité	134
États accessibles	134
Automate émondé	135
Complémentaire	135
Produit de machines	136
Intersection et union	137
Automate déterministe incomplet	139
Mots bloquants	139
Co-accessibilité	139

Complétion	140
Automate fini non-déterministe	141
Calcul	142
Langage associé	142
Automate des parties	143
Automate de Glushkov	145
Théorème de Glushkov	146
Transport des expressions régulières	146
Transport des automates	147
Tout langage rationnel est reconnaissable	148
Langages partiels d'ordre k	157
Rationalité des langages reconnaissables	158
Lemme de l'étoile ou Pumping lemma	160
Langage dérivé	161
Premières propriétés	161
Critère de rationalité	163
Morphismes d'automates	163
Automates équivalents	164
Équivalence de sommets	165
Équivalences restreintes	166