Lycée

aidherbe

Cours d'informatique : option

Liste des chapitres

Chapitre I

A.B.R.

1 Dictionnaires et ensembles

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 on suppose donnée une fonction **injective** `cle : 'a -> int` qui associe un indice unique à chaque élément.
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.
Nous allons dans ce chapitre définir une structure concrète qui permettra d'implémenter les dictionnaires de manière un peu moins efficace, la complexité sera logarithmique, mais qui sera persistante et qui permet une recherche d'intervalles.
Pour cela nous allons 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 les exemples les éléments seront des couples (a, b) où a est la clé : on recherche une valeur b associée à a.

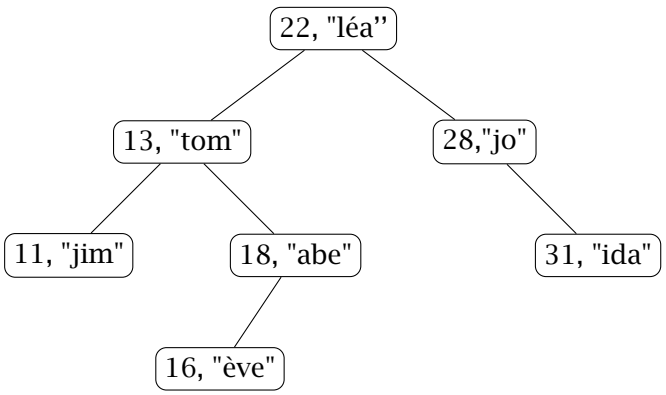
2 Arbres binaires de recherche

2-1 Définition

Arbre binaire de recherche (ABR)

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

Exemple



Dans la suite les représentations graphiques ne contiendront que la clé.
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*int*'a*'a abr);;
```

2-2 Recherche

Une fonction de base est la recherche d'un élément dans un arbre : on renvoie un booléen indiquant si l'élément dont la clé est donnée appartient 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 du nœud.

↪ C'est le principe de la recherche par dichotomie.
↪ On parcourt ainsi (partiellement) une branche de l'arbre.
↪ La complexité maximale est la hauteur de l'arbre.

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

Code I.1 Recherche dans un arbre binaire de recherche

Analyse

Nous allons établir la terminaison, la preuve et la complexité en même temps en séparant les cas où l'élément recherché est ou n'est pas dans l'arbre.

↪ 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).
↪ En s'inspirant du résultat ci-dessus on note $\mathscr{P}_h$ la propriété :
pour tout arbre `a` *de hauteur h et pour tout 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 correspondant à la clé recherchée ainsi $h \geqslant 0$.
1. $\mathscr{P}_0$ est vraie car un arbre de hauteur 0 qui contient la clé `k` ne peut être que l'arbre

`Noeud(Vide, k, x, Vide)` : dans ce cas la fonction renvoie `true` après 1 comparaison.

2. On suppose que $\mathscr{P}_m$ est vraie pour tout $m < h$ ($h \geqslant 1$).
   `a = Noeud(g, p, x, d)` est un arbre de hauteur h qui contient la clé `k`.

   i. Si on a `k = p`, la fonction `chercher k a` renvoie `true` avec 1 comparaison.

   ii. Si on a `k < p` alors aucun nœud de `d` n'a la clé `k` donc `g` contient la clé `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 `p < 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 $\mathscr{O}(h)$.

2-3 Lecture

La valeur d'un élément en fonction de sa clé est une modification simple du programme précédent.

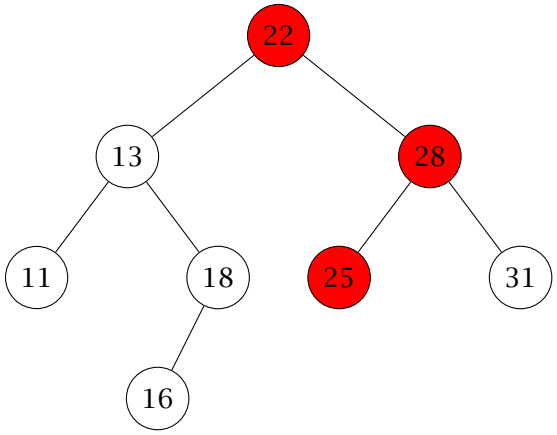
```
let rec valeur k arbre =
  match arbre with
   |Vide -> failwith "Il n'y a pas d'élément associé"
   |Noeud(g, n, x, d) when n = k -> x
   |Noeud(g, n, x, d) when k < n -> valeur k g
   |Noeud(g, n, x, d) -> valeur k d;;
```

Code I.2 Valeur associée à une clé

2-4 Insertion aux feuilles

On peut ajouter un nouvel élément à une feuille de l'arbre : on recherche l'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 :



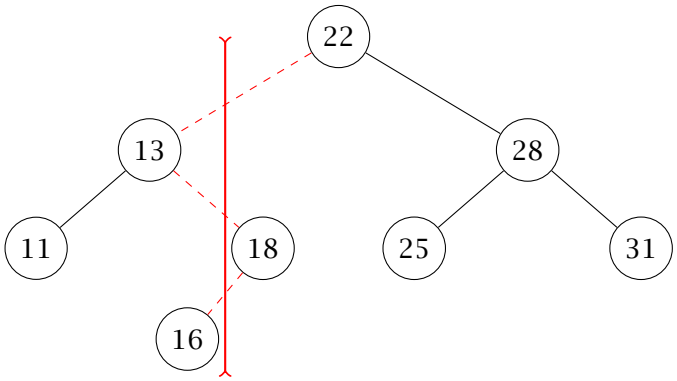

```
let rec insertionFeuille k x arbre =
  match arbre with
  |Vide -> Noeud(Vide, k, x, Vide)
  |Noeud(g, n, y, d) when n = k-> Noeud(g, n, x, d)
  |Noeud(g, n, y, d) when k < n
                  -> Noeud((insertionFeuille k x g), n, y, d)
  |Noeud(g, n, y, d) -> Noeud(g, n, y, (insertionFeuille k x d));;
```

Code I.3 Insertion à une feuille dans un arbre binaire de recherche

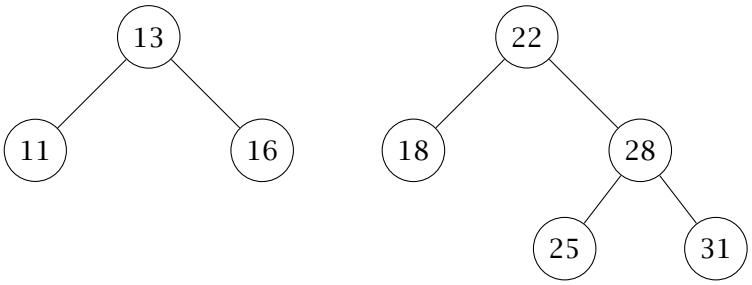
On remarquera que la valeur est changée si la clé existait déjà : on s'interdit les clés multiples.

2-5 Insertion à la racine

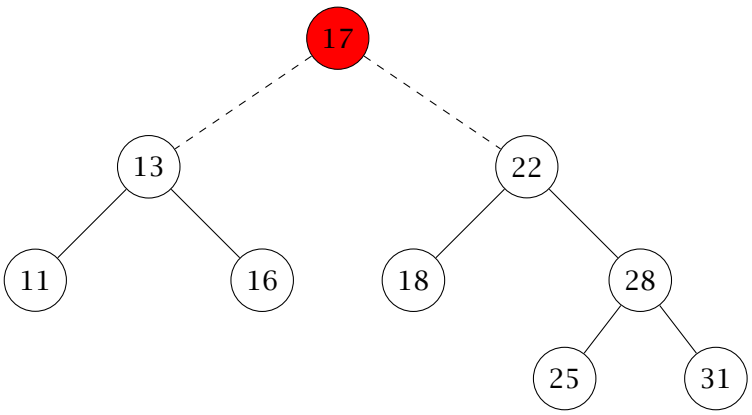
On peut aussi choisir d'insérer un nouveau nœud en le plaçant à la racine.
Pour cela on va couper l'arbre en deux parties regroupant les nœuds inférieurs à la nouvelle clé et ceux qui sont supérieurs. La récursivité permet de le faire avec un coût minimum c'est-à-dire proportionnel à la hauteur et non à la taille.
Le dernier arbre construit est coupé par 17 :



On reconstruit deux arbres :

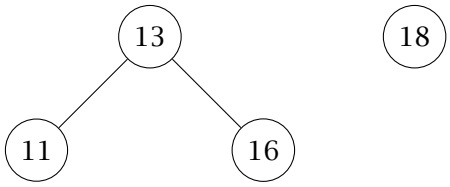


On peut alors ajouter le nœud 17 à la racine.



La première étape consiste à produire les deux arbres séparés par la valeur du nœud que l'on veut ajouter. Le découpage et la reconstitution se font en même temps, récursivement.
On suppose que la clé de coupure est strictement inférieure à la clé de la racine.

↪ On découpe le fils gauche avec la même clé de coupure. 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 clé de coupure est strictement supérieure à la clé de la racine.
En cas d'égalité on retire la racine et on renvoie les deux fils. Globalement cela aura pour effet de placer cette valeur de la clé à la racine.

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

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

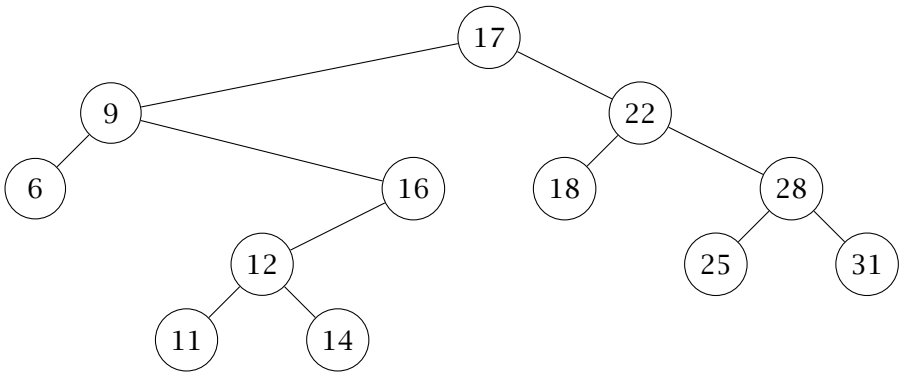
Code I.4 Insertion à la racine dans un arbre binaire de recherche

2-6 Suppression

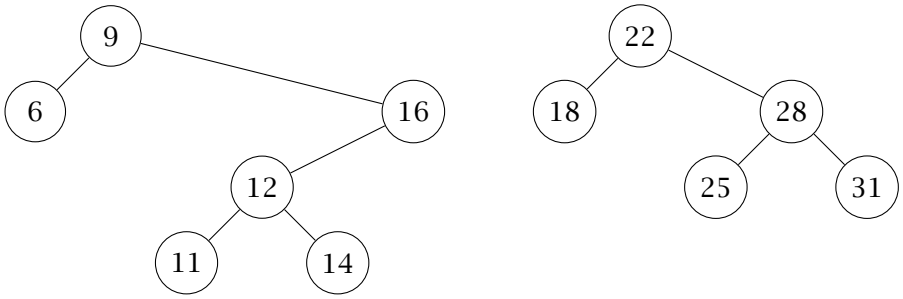
La suppression d'un nœud demande de conserver la structure ordonnée.
Il est relativement facile d'imaginer comment supprimer une feuille : on la remplace par l'arbre vide.
Le cas d'un nœud interne n est plus délicat. On se ramène au cas de la racine car il suffira de remplacer le sous-arbre dont n est la racine par l'arbre diminué.
Si on ôte la racine d'un sous-arbre il faut choisir un nœud à placer à la racine. Pour le faire sans trop de modification on va choisir un nœud du fils gauche et la placer à la racine. La

valeur de sa clé sera inférieure à toutes les clés du fils droit et pour majorer les clés du fils gauche il faut choisir le maximum dans le fils gauche.
On peut aussi choisir le minimum du fils droit.
Si la racine n'avait pas de fils gauche le nouvel arbre serait simplement le fils droit.

On va donc opérer en 4 étapes.

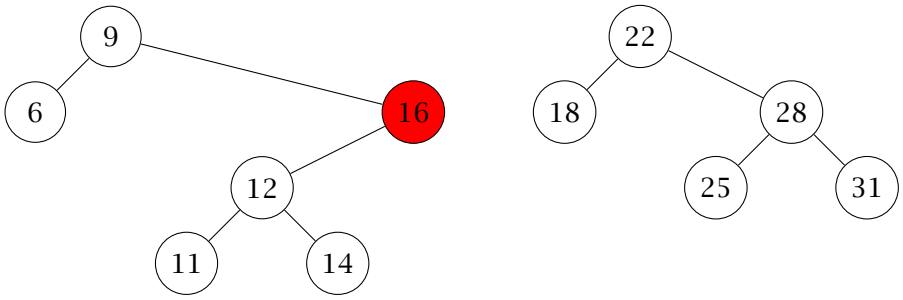


Enlever la racine



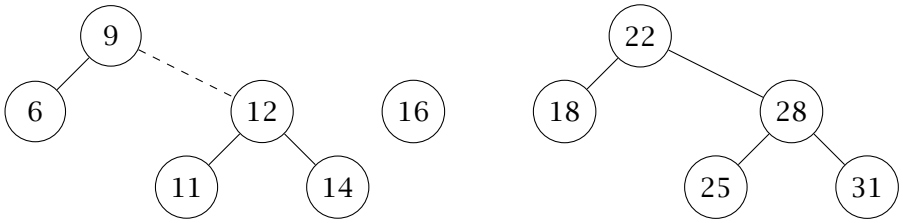
Calculer la clé maximale du fils gauche

On cherche récursivement le fils droit sans fils droit.

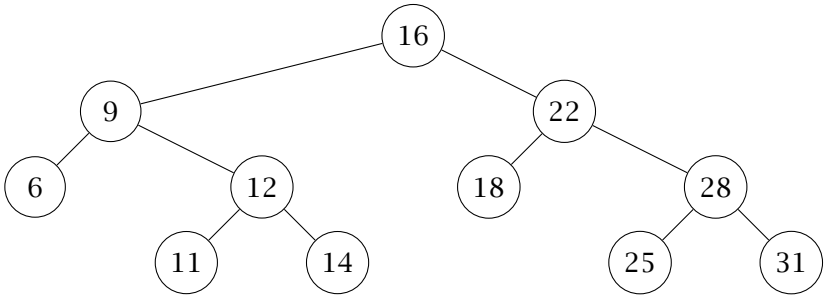


Reconstituer le fils gauche

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 -> raise(Failure "Arbre vide")
  |Noeud(_, n, x, Vide) -> n, x
  |Noeud(_, _, _, d) -> maxArbre d;;
```

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

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

Code I.5 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, n, x, d) when n = k
                      -> suppressionRacine arbre
  |Noeud(g, n, x, d) when k < n
                      -> Noeud((suppression k g), n, x, d)
  |Noeud(g, n, x, d) -> Noeud(g, n, x, (suppression k d));;
```

Code I.6 Suppression d'un nœud dans un arbre binaire de recherche

3 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 son les clés de `arbre` comprises entre `a` et `b`.

Ex. 5 **Tri**

Expliquer comment on peut 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 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 son prédécesseur n'a pas de fils droit (resp. son successeur n'a pas de fils gauche).

2. Montrer que si a'' est le successeur de a' alors
soit a'' 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 a''.

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. 8 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 = Noeud(g,k,d)` et `t1 = Noeud(g1,k,d1)` avec `g` prefixe de `g1` et `d` prefixe de `d1`.

1. Montrer que la relation "*est prefixe 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. 9 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.

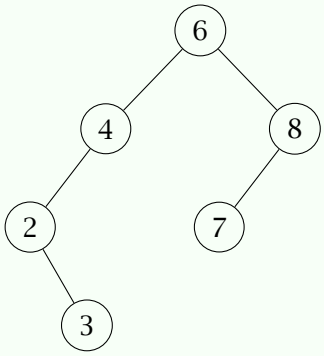
3. n_1 et n_2 sont deux nœuds d'un arbre. Prouver que leur PPAC est

   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. 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 $a_{i,n} = (a_{i-1} + 2)\dfrac{i-1}{n} + \dfrac{1}{n} + (a_{n-i} + 2)\dfrac{n-i}{n}$.
3. En déduire que $a_n = 2 - \dfrac{1}{n} + \dfrac{2}{n^2}\displaystyle\sum_{i=1}^{n-1} i a_i$
4. En calculant $n^2 a_n - (n-1)^2 a_{n-1}$ prouver que $a_n = \dfrac{1}{n^2}((n^2-1)a_{n-1} + 4n - 3)$
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

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

```
tasVide : '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>
```

↪ `tasVide` 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 unestructure 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.

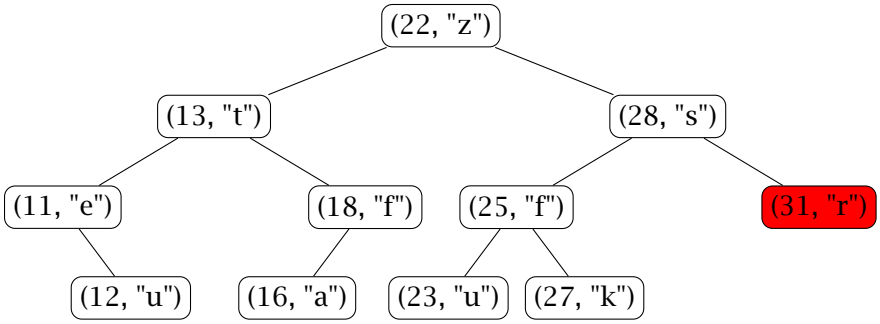
```
type 'a fdp = {mutable contenu : 'a abr};;

let cle = priorite;;
```

```
let tasVide 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.

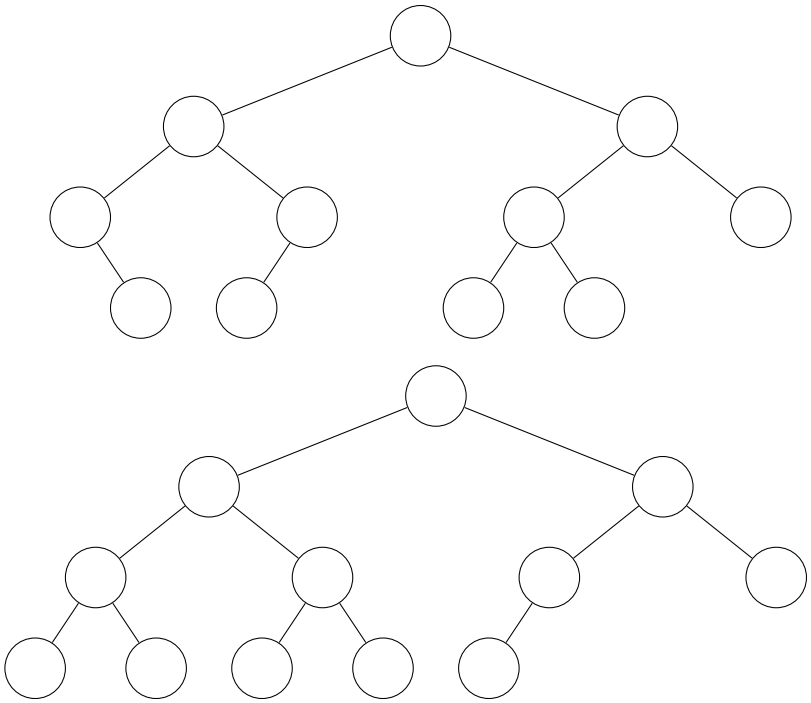
2-1 Définitions

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.

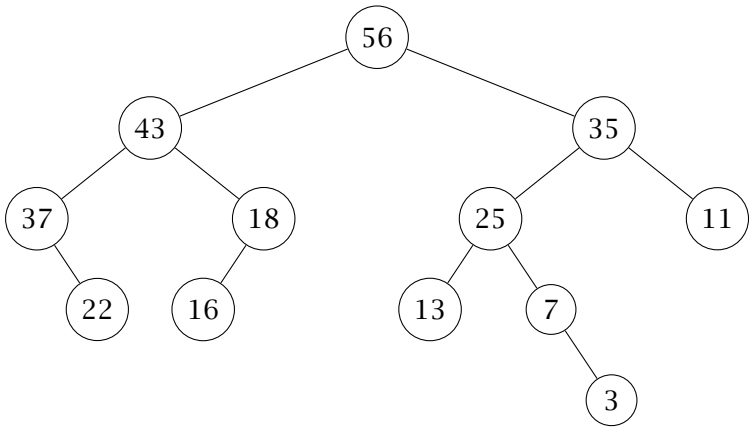
On a vu qu'alors le nombre de nœuds de profondeur k est 2^k pour tout $k < h$.
On en déduit que la taille n de l'arbre vérifie $2^{h-1} \leqslant n < 2^h$.



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.

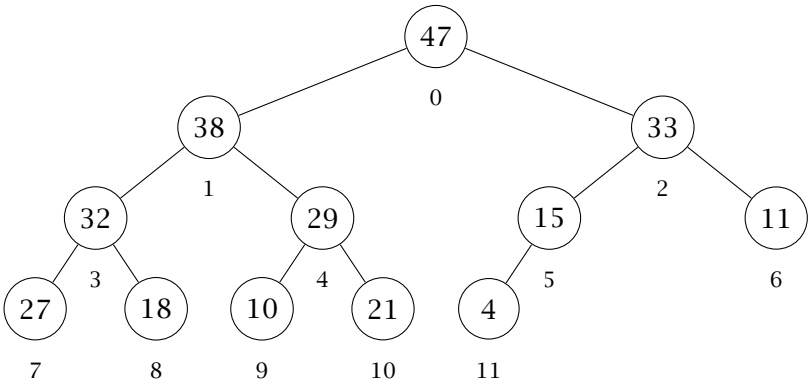
On a donc remplacé la croissance des clé "en largeur" des arbres binaires de recherche par une croissance du bas vers le haut moins contraignante.



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.

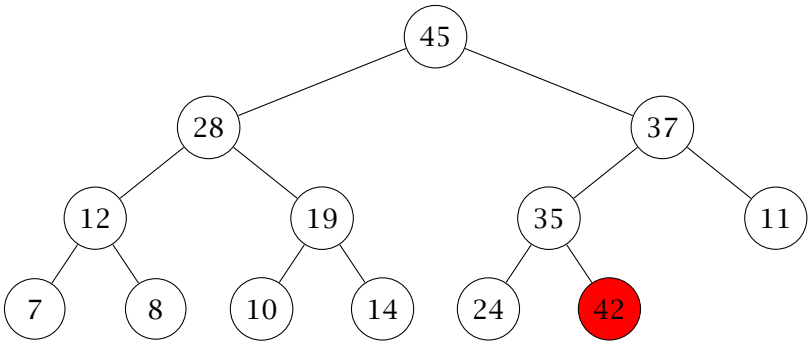


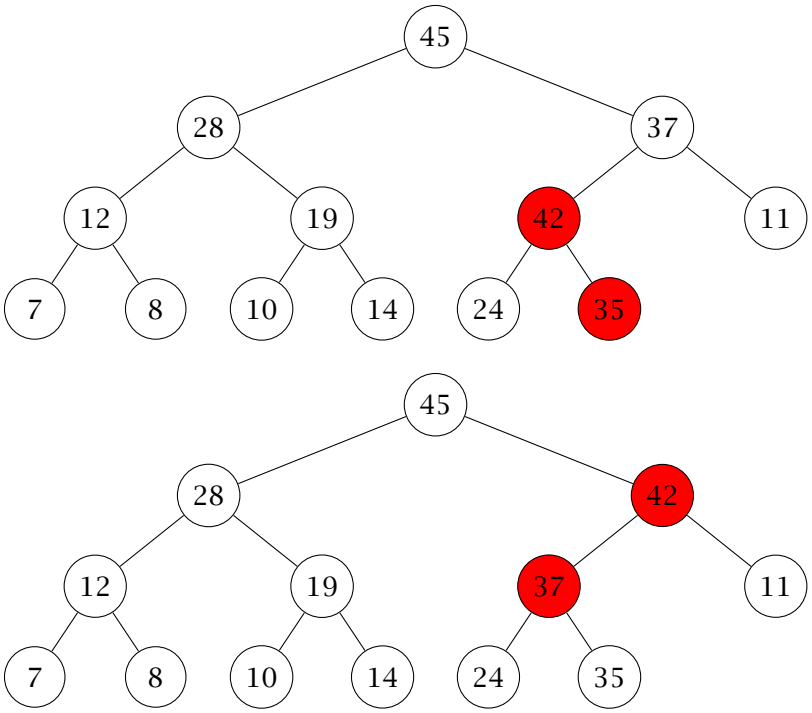
2-2 Opérations abstraites

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.

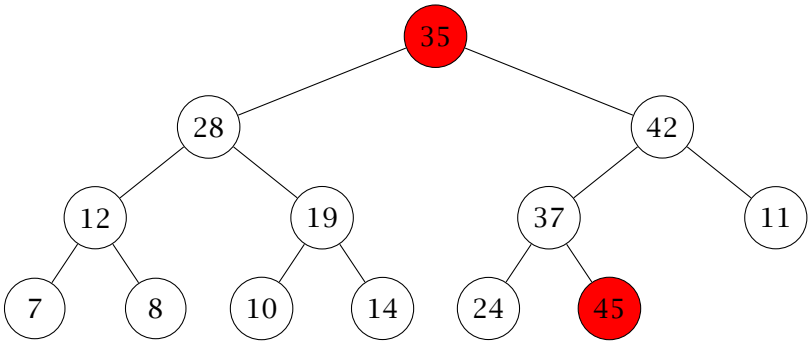


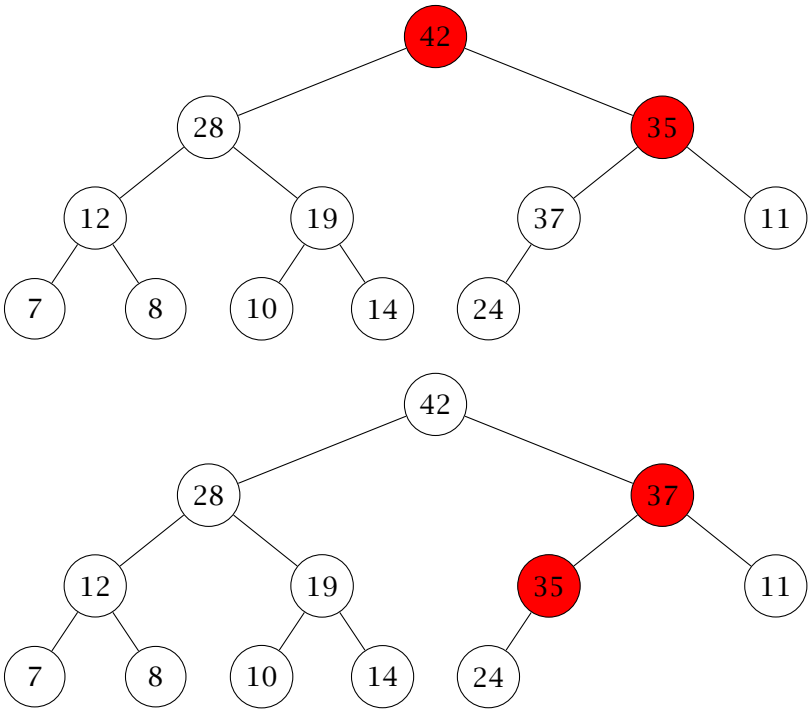


Enlever

Pour ôter la racine on peut

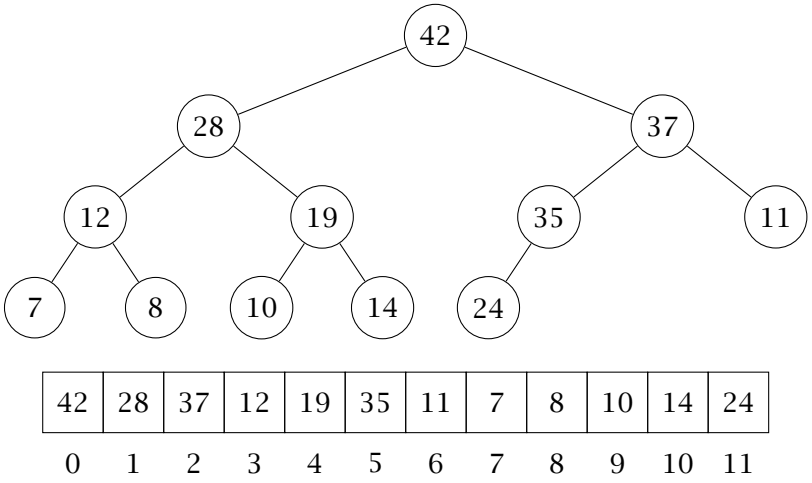
- ↪ échanger la racine avec le dernier élément,
- ↪ enlever le dernier élément
- ↪ à partir de la racine, échanger l'élément de la racine avec le plus grand de ses fils jusqu'à ce qu'on obtienne un tas




42
28
35
12
19
37
11
7
8
10
14
24
42
28
37
12
19
35
11
7
8
10
14
24

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 arrivern'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 tasVide 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 redescendre 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);;


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.2 Retrait tu 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.

2-4 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 $n-1$

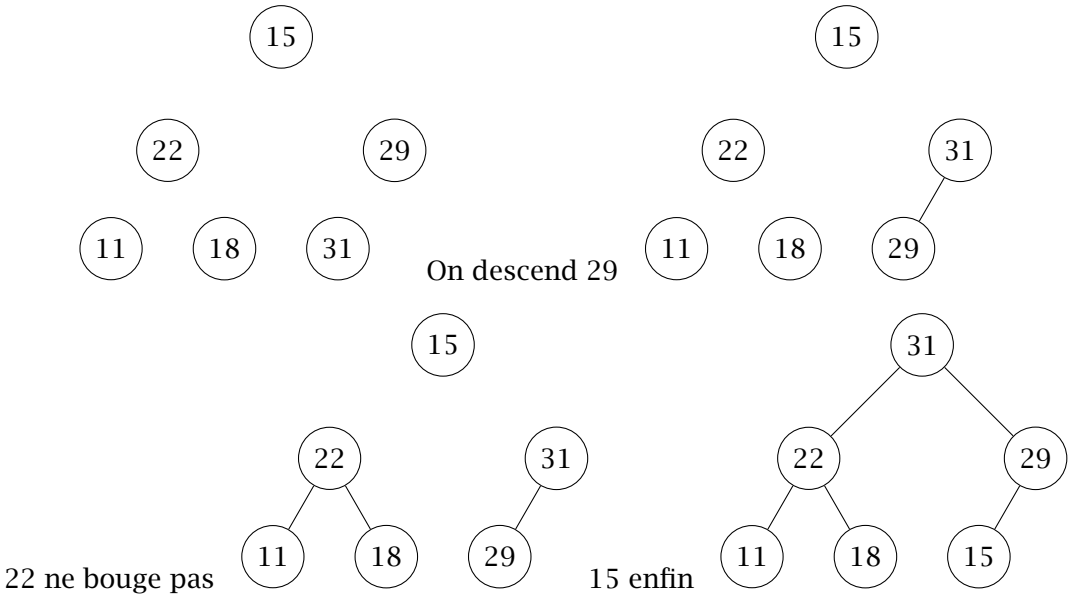
Ex. 1 **Création d'un tas** En utilisant uniquement la fonction `remonter`, écrire une fonction qui transforme `t` en un tas.
Quelle en est la complexité ?

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. 2 **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 liaison qui font un tas



Ex. 3 **Une amélioration**
Écrire une fonction qui transforme `t` en un tas avec cette méthode
Quelle est sa complexité ?

3 Exercices

Ex. 4 **Test de tas** Écrire une fonction qui teste si un tableau correspond à un tas.

Ex. 5 **Suppression dans un tas** Écrire une fonction qui supprime, dans un tas, l'élément d'indice k.

3-1 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 supposerons 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 supposerons, pour simplifier, que les indices sont des entiers majorés **a-priori** par une valeur `nMax` qui sera définie comme une variable globale. On gérera 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.

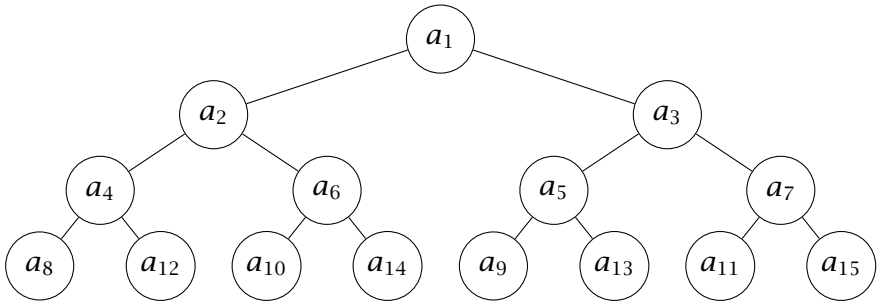
Ex. 6 **Tas modifiables**
Modifier les fonctions définies dans le cours et ajouter des fonctions
`present index tas` qui teste si un indice est déjà présent et
`changerPrio index valeur tas` qui modifie la clé de priorité avec la valeur indiquée.

3-2 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.

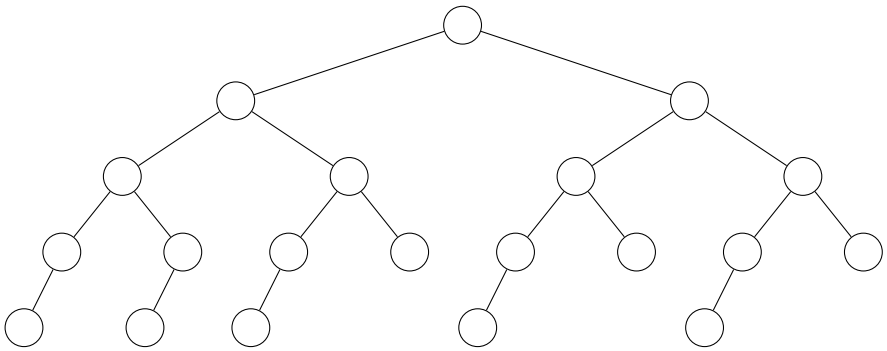
Ex. 7 Tableaux fonctionnels

1. Écrire une fonction `change k x a` qui renvoie une 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 `creeTableau 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 = Noeud(g,x,d)` est un arbre de Braun.

1. Montrer que `g` et `d` sont des arbres de Braun.
2. Montrer que `taille g` = `taille d` ou `taille g` = `taille d` +1.
3. Montrer que `hauteur g` = `hauteur d` ou `hauteur g` = `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 `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 ?

3-3 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

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 Syntaxe et sémantique

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".

1-1 Syntaxe

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 $\mathscr{A}$ cet ensemble ; ses éléments pourront être notés p, q, r, ...ou p_0, p_1, p_2 ...
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 : $w = x_1x_2\cdots x_m$.
Si $\mathscr{A}$ est un ensemble de variables propositionnelles, on note $\mathscr{A}' = \mathscr{A} \cup \{(,), \neg, \wedge, \vee\}$.
Les ensembles considérés sont des sous-ensembles de $(\mathscr{A}')^*$.

Formules

L'ensemble des formules de la logique des propositions sur $\mathscr{A}$ est le plus petit ensemble $\mathscr{L}$ (ou $\mathscr{L}_{\mathscr{A}}$) tel que
1. toute variable propositionnelle appartient à $\mathscr{L}$,
2. si $F \in \mathscr{L}$ est une formule, alors $(\neg F) \in \mathscr{L}$,
3. si $F \in \mathscr{L}$ et $G \in \mathscr{L}$ alors $(F \wedge G) \in \mathscr{L}$,
4. si $F \in \mathscr{L}$ et $G \in \mathscr{L}$ alors $(F \vee G) \in \mathscr{L}$.

Cette définition d'apparence anodine contient plusieurs présupposés :

a. il existe au moins un ensemble qui vérifie ces propriétés,
b. "le plus petit ensemble" a un sens,
c. il existe un plus petit ensemble.

Voici comment le sens que l'on peut donner à ces phrases.

a. Un ensemble existe : $(\mathscr{A}')^*$ est un exemple.

b. Un plus petit ensemble est défini par la relation d'inclusion. l'unicité d'un éventuel plus petit ensemble est alors acquise car, si L_1 et L_2 sont inclus dans tous les ensembles qui vérifient les propriétés, alors $L_1 \subset L_2$ et $L_2 \subset L_1$ donc $L_1 = L_2$.

c. On note $\mathbb{E}$ l'ensemble des ensembles qui vérifient les propriétés 1. à 4.
On définit $\mathscr{L} = \bigcap_{E \in \mathbb{E}} E$: montrons que $\mathscr{L}$ répond à toutes nos exigences.

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

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

Définition par induction

Cette construction de $\mathscr{L}$ ne permet pas facilement de reconnaître les formules.
Voici une autre manière de définir l'ensemble des formules.
↪ On défini $\mathscr{L}_0 = \mathscr{A}$.
↪ Si $\mathscr{L}_n$ est définit on construit $\mathscr{L}_{n+1}$ par

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

Définition inductive

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

Démonstration *On note $\mathscr{L}' = \bigcup_{n\in\mathbb{N}} \mathscr{L}_n$.*
On remarque que $\mathscr{L}_n \subset \mathscr{L}_{n+1}$ donc $\mathscr{L}_p \subset \mathscr{L}_q$ pour $p \leqslant q$.

1. *On a $\mathscr{A} = \mathscr{L}_0 \subset \mathscr{L}'$.*
2. *Si F appartient à $\mathscr{L}'$ alors il existe $n \in \mathbb{N}$ tel que $F \in \mathscr{L}_n$.*
   On a alors $\neg F \in \mathscr{L}_{n+1}$ d'où $\neg F \in \mathscr{L}'$.
3. *Si F et G appartient à $\mathscr{L}'$ alors il existe $p, q \in \mathbb{N}$ tel que $F \in \mathscr{L}_p$ et $F \in \mathscr{L}_q$.*
   Si on pose $n = \max\{p, q\}$, on a $F \in \mathscr{L}_p \subset \mathscr{L}_n$ et $G \in \mathscr{L}_q \subset \mathscr{L}_n$.
   On en déduit que $(F \wedge G) \in \mathscr{L}_{n+1} \subset \mathscr{L}'$.
4. *De même $(F \wedge G) \in \mathscr{L}'$ pour F et G dans $\mathscr{L}'$.*

Ainsi $\mathscr{L}'$ vérifie les propriétés 1. à 4.

Soit E vérifiant les propriétés 1. à 4.
On montre par récurrence sur n que alors $\mathscr{L}_n \subset E$ pour tout n.
↪ $\mathscr{L}_0 = \mathscr{A} \subset E$.
↪ *Si $\mathscr{L}_n \subset E$ alors*
pour tout $F \in \mathscr{L}_n$, on a $F \in E$ donc $\neg F \in E$: $\{\neg F,\ F \in \mathscr{L}_n\} \subset E$,
pour tout $F, G \in \mathscr{L}_n \subset E$ on a $(F \wedge G) \in E \in E$: $\{F \wedge G,\ F, G \in \mathscr{L}_n\} \subset E$,
de même $\{F \vee G,\ F, G \in \mathscr{L}_n\} \subset E$.
Ainsi l'union des ensembles, $\mathscr{L}_{n+1}$, est incluse dans E.
↪ *La récurrence montre bien que $\mathscr{L}_n \subset E$ pour tout n.*

On en déduit que $\mathscr{L}' = \bigcup_{n\in\mathbb{N}} \mathscr{L}_n \subset E$.
$\mathscr{L}'$ vérifie les 4 propriétés et est inclus dans tout ensemble de $\mathbb{E}$,
c'est le plus petit ensemble vérifiant les propriétés 1. à 4. d'où $\mathscr{L}' = \mathscr{L}$.

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 $\mathscr{L}$ 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 $f(G)$.

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.

Arbre syntactique

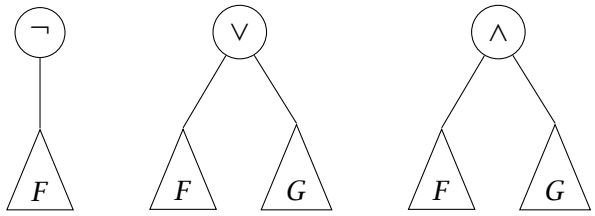
On définit par induction structurelle un arbre à partir d'une formule.

↪ Une variable propositionnelle x est représentée par 

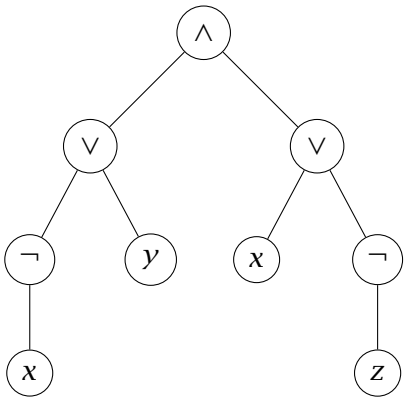
↪ Si F est représentée par 

et G par 

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;;
```

1-2 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 $\mathscr{A}$, l'ensemble des variables propositionnelles, vers $\mathscr{B} = \{\mathbf{V}, \mathbf{F}\}$.

La sémantique consiste à prolonger une valuation ν en une application $\tilde{\nu}$ de l'ensemble des formules, $\mathscr{L}$, vers l'ensemble des valeurs de vérité, $\mathscr{B}$. 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 $\tilde{\nu}$ par induction structurelle :
↪ si $x \in \mathscr{A}$ alors $\tilde{\nu}(x) = \nu(x)$,
↪ si $F, G \in \mathscr{L}$ alors $\tilde{\nu}(\neg F)$, $\tilde{\nu}(F \wedge G)$ et $\tilde{\nu}(F \vee G)$ sont définis par les tables de vérité.

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

Formules satisfaites

ν désigne une valuation, F désigne une formule.
↪ Si $\tilde{\nu}(F)$ est **V** on dit que ν **satisfait** F et que F **est satisfaite** par ν.
↪ 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 $\tilde{\nu}(F)$.
Exemple : $F = (\neg x \vee y) \wedge (x \vee \neg z)$

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

2 Équivalence

2-1 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 structurelle 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 $\tilde{\nu}$ *:* $\tilde{\nu}(\neg G)$, $\tilde{\nu}(G \vee H)$ *et* $\tilde{\nu}(G \wedge H)$ *ne dépendent que de* $\tilde{\nu}(G)$ *et* $\tilde{\nu}(H)$ *qui sont égaux respectivement à* $\tilde{\nu}(G')$ *et à* $\tilde{\nu}(H')$.

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

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

2-2 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 $p_1 \vee \ldots \vee p_n$ où $n \geq 1$ et $p_1, \ldots, p_n$ sont des littéraux.
- Une **clause conjonctive** est une conjonction de littéraux.
  C'est une formule de la forme $p_1 \wedge \ldots \wedge p_n$ où $n \geq 1$ et $p_1, \ldots, p_n$ sont des littéraux.
- Une **forme normale conjonctive** est conjonction de clause.
  C'est une formule de la forme $C_1 \wedge \ldots \wedge C_m$ où $m \geq 1$ et $C_1, \ldots, C_m$ sont des clauses.
- Une **forme normale disjonctive** est disjonction de clauses conjonctives : $D_1 \vee \ldots \vee D_m$.

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 la table de vérité de F.

↪ On considère chaque ligne qui donne un résultat **V**.
Elle est associée à une valuation ν.
↪ À chaque variable x on associe le littéral x si $\nu(x) =$ **V** ou $\neg x$ si $\nu(x) =$ **F**.
↪ On associe alors la formule D_ν conjonction de ces littéraux

Forme normale canonique

La disjonction des formules D_ν 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 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

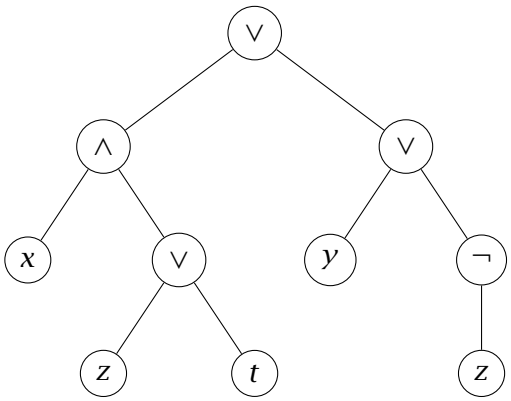
$$(\neg x \vee y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (x \vee \vee \neg y \neg z) \wedge (x \vee \vee y \neg z)$$

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

3 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 $x, y \in \mathscr{A}$ 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
- $\neg(\neg F) \equiv F$
- Si G est une tautologie alors $F \wedge G \equiv F$
- Si G est une contradiction alors $F \vee G \equiv F$

Ex. 5 Prouver les lois de Morgan
- $\neg(F \wedge G) \equiv \neg F \vee \neg G$
- $\neg(F \vee G) \equiv \neg F \wedge \neg G$

Ex. 6 Prouver les lois de distributivité
↪ $F \wedge (G \vee H) \equiv F \wedge G \vee F \wedge H$
↪ $F \vee G \wedge H \equiv (F \vee G) \wedge (F \vee H)$

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, ..., x_n.
V_F est l'ensemble des valuations ν des n variables telles que $\nu(F) = \mathbf{V}$.
Pour une valuation ν et pour chaque variable x on définit le littéral
$l_\nu(x) = x$ $\nu(x) = \mathbf{V}$ et $l_\nu(x) = \neg x$ si $\nu(x) = \mathbf{F}$.
On pose alors $D_\nu = \bigwedge_{i=1}^{n} l_\nu(x_i)$ et $F_N = \bigvee_{\nu \in V_F} D_\nu$.
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 $\mathbf{ite}(F, G, H)$ est équivalente à $(F \wedge G) \vee (\neg F \wedge H)$.

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

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 $\tilde{\nu}(\mathbf{0}) = \mathbf{F}$ et $\tilde{\nu}(\mathbf{1}) = \mathbf{V}$ pour toute valuation ν.
   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 $[F \Leftrightarrow G] \equiv [F \wedge G \vee \neg F \wedge \neg F]$.
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

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 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

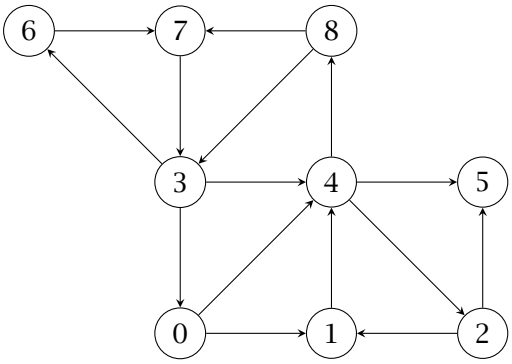
Graphe

Un graphe orienté est un couple (S, A) où
↪ S est un ensemble fini de **sommets** ou **nœuds**,
↪ 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).*

$$\begin{aligned} \text{Exemple : } S = &\{0,1,2,3,4,5,6,7,8\} \\ A = &\{(0,1),(0,4),(1,4),(2,1),(2,5), \\ &(3,0),(3,4),(3,6),(3,8),(4,2), \\ &(4,5),(4,8),(6,7),(7,3),(8,7)\} \end{aligned}$$

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

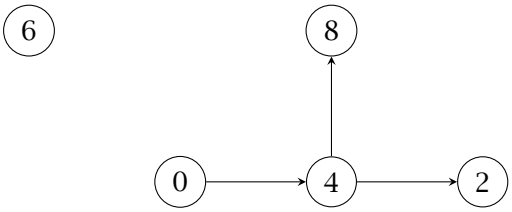


Le plus souvent l'ensemble des sommets sera $\{0, 1, 2, ..., n-1\}$.
Le cardinal de S, $|S|$, est la taille du graphe.
$|A|$ donne une idée de la densité du graphe, c'est un entier compris entre 0 et $|S|(|S|-1)$.

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.

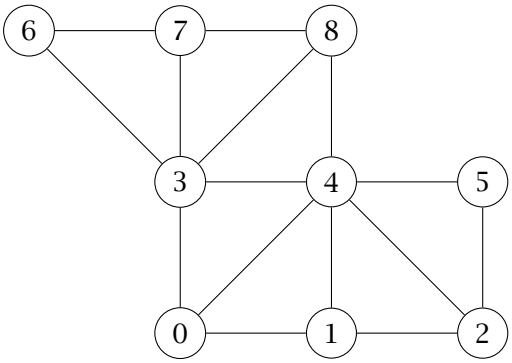
1-1 Graphes non orientés

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 at a.
On simplifiera la donnée du graphe en considérant que les arêtes ne sont pas des couples mais des parties à deux éléments[1] de S.

Graphe non oriente

Un **graphe** non orienté est un couple (S, A) où
↪ S est un ensemble fini,
↪ A est une partie de $\mathscr{P}_2(S)$ 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

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

On le notera aussi $a \to a_1 \to a_2 \to \cdots \to a_{k-1} \to b$.
↪ 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

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

avec $n \geqslant 3$ 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 $\{0, 1, 2, 3, 4, 6, 7, 8\}$ 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 mutables : 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 peu 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]; [4]; [1; 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 ajouteO s liste =
    match liste with
    |[]   -> []
    |t::q -> (s, t)::(ajouteO s q) in
  for i = 0 to (n -1) do
    ar := (ajouteO 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 $d \leqslant n = |S|$.

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 différentient 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

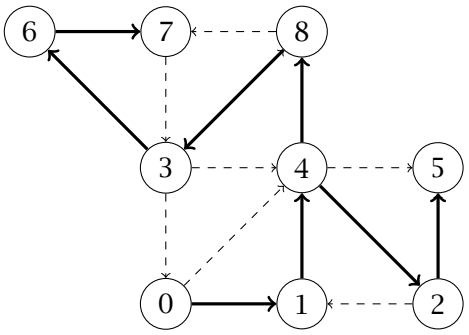
Le parcours récursif consiste à marquer un sommet comme vu puis visiter récursivement ses voisins non encore vus. On peut tester si le sommet a été vu avant ou après l'appel récursif. On choisit ici de le faire immédiatement lors du traitement du sommet.

```
let parcours g s0 =
  let vus = Array.make (taille g) false in
  let rec visiter s =
    if not vus.(s)
    then (vus.(s) <- true;
          List.iter visiter (voisins g s))
  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
4	`[V, V, F, F, V, F, F, F, F]`	1
2	`[V, V, V, F, V, F, F, F, F]`	4
5	`[V, V, V, F, V, V, F, F, F]`	2
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 au plus qu'un appel de la fonction par arête du graphe : 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 : $s = s_0 \to s_1 \to \cdots \to s_{k-1} \to s_k = t$.
   s_0 est marqué comme vu et t ne l'est pas : il existe donc $i \in \{1, 2, \ldots, k\}$ 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 valida 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 largeur

On peut écrire des parcours sans utiliser la récursivité comme ci-dessus. On peut alors maîtriser l'ordre de parcours des sommets. Dans ce but on ne visite pas directement les voisins mais on les place en attente.
L'idée du parcours en largeur est de visiter d'abord tous les voisins de s_0 puis les voisins de ces voisins et ainsi de suite. On veut donc que la mise en attente fassent sortir les sommets dans l'ordre d'entrée : c'est la structure de `file`.

Les étapes sont donc

1. On crée une file vide E et un tableau de booléens initialisés à FAUX.
2. On place le sommet initial dans E.
3. Tant que la file E est non vide :
   1. on prend le premier sommet et s'il n'a pas encore été vu
   2. on le marque comme vu
   3. on le place dans la file.

Le type de données `file` a été vu en première année.
On va utiliser les files du module **`Queue`** de **`OCaml`**:

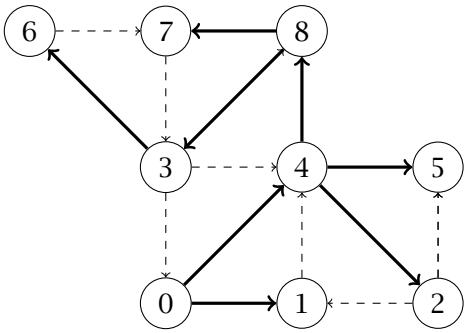
↪ le type sera noté `type 'a t` où `` `a `` le type des données dans la file.
Les files sont mutables, elles sont modifiées sans nouvelle création.
↪ `Queue.create : unit -> 'a t` crée une file vide,
↪ `Queue.is_empty 'a t -> bool` teste si une file est vide,
↪ `Queue.add 'a -> 'a t -> unit` ajoute un 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 vus = Array.make (taille g) false in
  let attente = Queue.create() in
  let traiter s = if not vus.(s)
                  then (Queue.add s attente;
                        vus.(s) <- true) in
  Queue.add s0 attente;
  vus.(s0) <- true;
  while not (Queue.is_empty attente) do
    let s = Queue.take attente in
    List.iter traiter (voisins g s) done;
  vus;;
```

Code IV.7 Parcours en largeur d'un graphe

Voici le parcours du graphe orienté donné en exemple. Les voisins sont renvoyés dans l'ordre croissant.

sommet traité	file	tableau vu	origine
initialisation	(0)	[V, F, F, F, F, F, F, F, F]	
0	(1, 4)	[V, V, F, F, V, F, F, F, F]	
1	(4)	[V, V, V, F, V, V, F, F, V]	0
4	(2, 5, 8)	[V, V, V, F, V, V, F, F, V]	0
2	(5, 8)	[V, V, V, F, V, V, F, F, V]	4
5	(8)	[V, V, V, F, V, V, F, F, V]	4
8	(7)	[V, V, V, F, V, V, F, V, V]	4
7	(3)	[V, V, V, V, V, V, F, V, V]	8
3	(6)	[V, V, V, V, V, V, V, V, V]	8
6	()	[V, V, V, V, V, V, V, V, V]	3



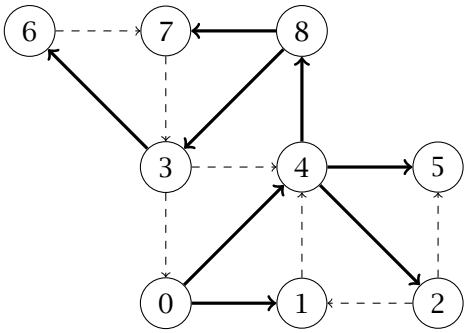
3-3 Parcours en profondeur

Si on remplace la file par une pile on obtient le parcours en profondeur.
On va utiliser les piles du **`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.8 Parcours en profondeur d'un graphe

On suppose ici que les voisins sont triés par ordre décroissant dans la liste afin d'obtenir un comportement semblable au parcours récursif.



sommet traité	pile	tableau vu	origine
initialisation	(0)	[V, F, F, F, F, F, F, F, F]	
0	(1, 4)	[V, V, F, F, V, F, F, F, F]	
1	(4)	[V, V, F, F, V, F, F, F, F]	0
4	(2, 5, 8)	[V, V, V, F, V, V, F, F, V]	0
2	(5, 8)	[V, V, V, F, V, V, F, F, V]	4
5	(8)	[V, V, V, F, V, V, F, F, V]	4
8	(3, 7)	[V, V, V, V, V, V, F, F, V]	4
3	(6, 7)	[V, V, V, V, V, V, V, F, V]	8
6	(7)	[V, V, V, V, V, V, V, V, V]	3
7	()	[V, V, V, V, V, V, V, V, V]	8

On voit, par exemple, le sommet 5 est vu depuis le sommet 4 plutôt que depuis le sommet 2 dans le parcours récursif. Pour obtenir un comportement semblable il suffit de marquer comme vus les sommet seulement au moment où on les sort de la pile.
On écrit une fonction qui reprend `Stack.push` en donnant les arguments dans l'autre sens afin de pouvoir l'appeler directement dans `List.iter`.

```
let ajouter pile x = Stack.push x pile;;

let parcoursProfondeur1 g s0 =
  let vus = Array.make (taille g) false in
  let attente = Stack.create() in
  Stack.push s0 attente;
  while not (Stack.is_empty attente) do
    let s = Stack.pop attente in
    if not vus.(s)
    then (vus.(s) <- true;
          List.iter (ajouter attente)  (voisins g s)) done;
  vus;;
```

Code IV.9 Parcours en profondeur modifié

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

Code IV.10 Calcul des composantes connexes

Dans les exercices cet algorithme sera aussi invoqué sous le nom de parcours récursif complet.
Sa complexité est toujours un $\mathcal{O}(|S| + |A|)$.

4-2 Plus court chemin

On peut souhaiter connaître le un chemin, lorsqu'il en existe, entre deux sommets.
Pour cela on peut remplacer le tableau de booléens par un tableau qui indique le sommet précédent lors de l'appel récursif. On peut alors construire un chemin sous la forme des sommets intermédiaires en "remontant" les antécédents depuis s_0.
On va le faire avec le parcours en largeur, on démontre ensuite qu'on obtient ainsi un chemin minimum en nombre d'arêtes.

On commence par calculer le tableau des antécédents.

```
let antecedents g s0 =
  let peres = Array.make (taille g) None in
  let attente = Queue.create() in
  let traiter orig dest = if peres.(dest) = None
                          then (Queue.add dest attente;
                                peres.(dest) <- Some orig) in
  Queue.add s0 attente;
  pere.(s0) <- Some s0;
  while not (Queue.is_empty attente) do
    let s = Queue.take attente in
    List.iter (traiter s) (voisins g s) done;
  peres;;
```

Code IV.11 Calcul des antécédents

On peut alors écrire la fonction demandée. Pour obtenir une liste des sommet dans le sens souhaité on écrit une fonction récursive terminale. On choisit de renvoyer une liste vide si les sommets ne sont pas connectés.

```
let chemin g s0 t0 =
  let peres = antecedents g s0 in
  let rec aux fin chemin =
      match peres.(fin) with
      |None -> []
      |Some u when u = s0 -> s0::chemin
      |Some u -> aux u (u::chemin)
  in aux t0 [t0] ;;
```

Code IV.12 Calcul du chemin entre deux sommets

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 $(a_1, a_2, \ldots, a_p)$ (dans l'ordre de leur insertion) alors il existe un entier d tel que
   1. $d = \delta(s_0, a_1) \leqslant \delta(s_0, a_2) \leqslant \cdots \delta(s_0, a_p) \leqslant d+1$,
   2. *tous les sommets s tels que $\delta(s_0, s) \leqslant d$ ont déjà été insérés dans la file*[3].

   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 $\delta(s_0, s) \leqslant d+1$.
   D'après l'hypothèse de récurrence, tous les sommets de distance d ou moins ont déjà été insérés donc on a $\delta(s_0, s) \geqslant d+1$.
   Ainsi $d(s_0, s) = d+1$ pour tous les sommets ajoutés, notés a_{p+1}, …, a_q.
   Deux cas sont possibles
   - ↪ *Soit on a $p \geqslant 2$ et $\delta(s_0, a_2) = \delta(s_0, a_1) = d$.*
     La suite devient $(a_2, \ldots, a_p, a_{p+1}, \ldots, a_q)$ 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 $p \geqslant 2$ et $d_2 = p+1$.*
     Dans ce cas la file peut être vide ou ne contient 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

     $$s_0 \to s_1 \to s_2 \to \cdots \to s_{d-1} \to s_d \to s_{d+1} = s$$

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

[3] Certains peuvent en être sortis

On considère le premier sommet traité parmi ceux (à distance d de s_0) qui ont 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*

$$s_0 \to s_1 \to s_2 \to \cdots \to s_{d-1} \to s_d = s$$

On sait que $d(s_0, s_k) = d(s_0, s_{k-1})$ 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 $|A| \geqslant |S| - 1$.

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

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 condition suivantes est réalisée.

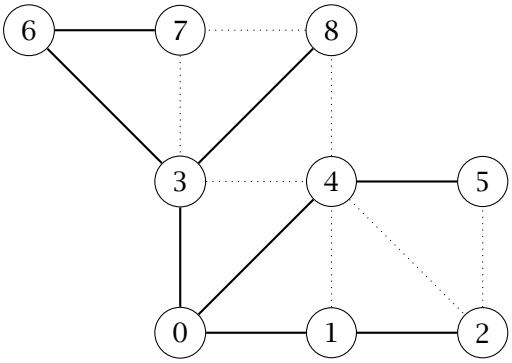
1. G est connexe et $|A| = |S| - 1$.
2. G est acyclique et $|A| = |S| - 1$.
3. G est connexe et n'est plus connexe si on retire une arête quelconque (toutes ses arêtes sont des isthmes).
4. G est acyclique et devient cyclique si on ajoute une arête.

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 longueur 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, ..., $k-1$ (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, ..., $k-1$ et k on peut
↪ ne passer que par les sommets 0, 1, ..., $k-1$
↪ 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, ..., $k-1$.

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 on 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 $\{0, 1, \ldots, n-1\}$ 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.13 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 encours 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.13**, **page 79**.

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

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 chemin et introduire un questionnement sur l'optimisation de ces valeurs :
↪ quel sont les chemins de poids minimum ?
↪ comment recouvrir à moindre coût le graphe ?
↪ quels est le flux maximum entre deux points ?
Cette dernière question, hors-programme, fera l'objet d'un chapitre de compléments.

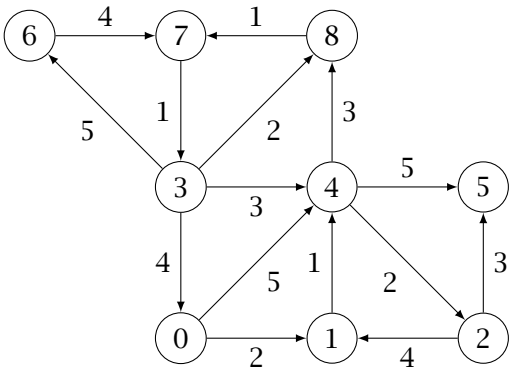
1 Introduction

1-1 Définitions

Graphe valué

Un graphe valué sur $\mathscr{B}$ est un triplet (S, A, w) où
↪ (S, A) est un graphe non orienté
↪ w est une application de A vers $\mathscr{B}$;
$w(i, j)$ 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 $c = ((a, a_1), (a_1, a_2), (a_2, a_3), \ldots, (a_{k-1}, b))$ est la somme des poids des arêtes qui le composent,
$W(c) = \sum_{i=1}^{k} w(a_{i-1}, a_i)$ avec $a_0 = a$ et $a_k = b$.

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

1-2 Implémentations

Les type 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 arc  = Infini|Poids of 'a;;

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

L'exemple ci-dessus est implémenté par

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

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

↪ On note $p_k(i,j)$ le poids minimum des chemin de $\Gamma_k(i,j)$.
Ainsi $p_n(i,j)$ est le poids minimum des chemin 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 $d(s_0,s) \geqslant w(s_0,s')$.
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, ..., s_{p-1}.
On cherche le suivant : s_p.

↪ On note $S' = S \setminus \{s_0, s_1, ..., s_{p-1}\}$: 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 $S \setminus S'$.Dijkstra
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 une distance possible à s_0. La clé de comparaison 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

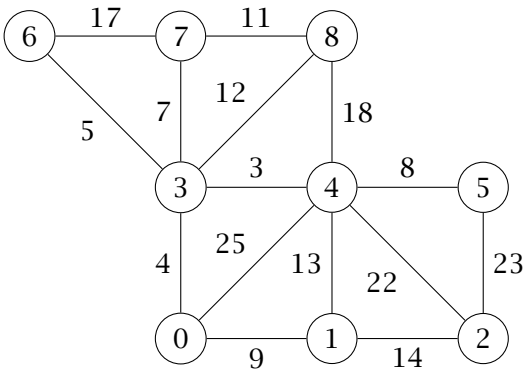
Complexité de l'algorithme de Dijkstra

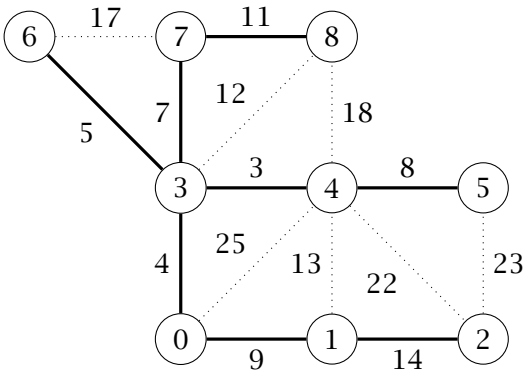
La complexité de l'algorithme est en $\mathcal{O}(p\log(p))$.

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 $p \leqslant n^2$ où n est le nombre de sommets on peut aussi majorer la complexité par un $\mathcal{O}(n^2\log(n))$.

3 Arbre couvrant minimum

Dans cette partie les graphes seront supposés **non orientés**.
On rappelle que les poids sont strictement positifs. Le parcours (récursif, en largeur ou en profondeur) depuis un sommet s_0 définit un ensemble d'arêtes qui formait un graphe connexe acyclique contenant tous les sommets de la composante connexe de s_0.
Dans le cas d'un graphe connexe on a donc un arbre rejoignant tous les sommets : on parle d'arbre couvrant.
Dans le cas d'un graphe valué le poids d'un arbre est la somme des poids des arcs le composant et il est naturel de rechercher un arbre couvrant de poids minimum.
L'ensemble des arbres couvrant est fini, ils correspondent à des parties à $n-1$ éléments de l'ensemble des arêtes. Un arbre couvrant minimum correspond au minimum d'un ensemble fini non vide : il existe au moins un arbre couvrant minimum.





3-1 Algorithme de Prim

Le premier algorithme, de Prim, est semblable à l'algorithme de Dijkstra :

```
On part d'un sommet s0
On ajoute les les arêtes une par une en choisissant celle
qui sort de l'arbre déjà construit qui est de poids minimum
```

Le seul changement est le tri par le poids de l'arête et non par le poids du chemin depuis s_0.

```
let prim g =
  let n = taille g in
  let vu = Array.make n false in
  let fp = creeVide (0, 0, 0) in (* On part du sommet 0 *)
  let acm = ref [] in
  while not (estVide fp) do
    let (s, t, d) = extraire fp in
    if not vu.(t)
    then begin
         vu.(t) <- true;
         acm := (s,t) :: !acm;
         List.iter (fun (u, w) -> ajouter (t, u, w) fp)
                   (voisins g t)
         end done;
 !acm;;
```

Code V.3 Algorithme de Prim

Coupures

Coupure

Une **coupure** d'un graphe (S, A) est une partition de S en deux partie non vides (et disjointes).
Une **arête traversante** associée à une coupure est une arête qui joint un sommet de la première partie à un sommet de la seconde partie.

Propriété de coupure

G est un graphe valué non orienté et connexe dont les poids sont distincts deux à deux.
Pour toute coupure il existe une unique arête traversante de poids minimal.
Tout arbre couvrant minimal contient alors cette arête.

Démonstration : l'unicité de l'arête traversante de poids minimal associée à une coupure découle du fait que les poids sont distincts.
On note a_0 l'arête de poids minimum associée à une coupure (S_1, S_2)..
On considère un arbre couvrant minimum t.
Si t ne contenait pas a_0 on obtiendrait un cycle en ajoutant a_0 à t : ce cycle doit contenir a_0.
Dans ce cycle il existe au moins une autre arête a traversante pour la coupure.
Si on ôte a (après avoir ajouté a_0) dans t on obtient encore un arbre couvrant t_1.
Le poids de t_1 est $w(t_1) = w(t) + w(a_0) - w(a) < w(t)$ en raison de la minimalité de $w(a_0)$ pour les arêtes traversantes.
On aboutit donc à une contradiction sur le poids minimal de t.
T doit contenir a_0.

Dans la suite les démonstration seront écrites dans le cas de graphes à poids distincts. On admettra qu'elles restent valides dans le cas général.
On démontrera en fait que, dans ce cas, il existe un seul arbre couvrant minimum t.

Preuve et complexité

L'algorithme de Prim fournit un arbre couvrant minimum.
Si on a utilisé un tas pour la file de priorité sa complexité est un $\mathcal{O}(m\ln(m))$ avec $m = |A|$.

Démonstration : à chaque étape on peut définir une coupure dont une partie est l'ensemble des sommets connectés.
L'algorithme ajoute alors une arête traversante de poids minimal associée à cette coupure : cette arête doit appartenir à tout arbre couvrant minimal.
On ajoute ainsi $n-1$ arêtes qui doivent toutes appartenir à tout arbre couvrant minimal.
Tout arbre couvrant minimal contient $n-1$ arêtes, ce sont celles qu'on vient de choisir.
On montre donc qu'on obtient bien un arbre couvrant minimal et que, de plus, celui-ci est unique.

Dans l'algorithme on insère toutes les arêtes dans une file d'attente et on peut toutes les extraire d'où la complexité.

3-2 Algorithme de Kruskal

Dans l'algorithme de Prim on ajoute des arêtes de poids minimal pour connecter un point de plus à chaque étape. Un autre algorithme, celui de Kruskal, a un principe dual : on utilise la propriété d'acyclicité des arbres plutôt que leur connexité.

```
On trie les arêtes selon leur poids croissant
On ajoute les arêtes une par une en choisissant celle de poids
minimum qui ne crée pas de cycle
```

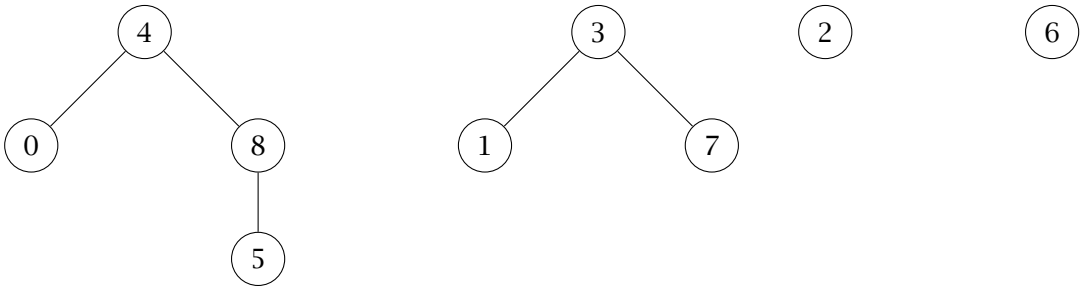
La difficulté est de repérer les arcs qui, ajoutés, forment un cycle.
Pour cela on va définir une structure de données qui permet de gérer les composantes connexes d'un graphe à sommets fixés dans lequel on ajoute des arêtes.
On commence par autant de composantes connexes que de sommets et on unifie deux composantes (distinctes) à chaque fois que l'on ajoute un arc qui les joint.

3-3 Type de données Union-Find

Dans ce but on définit le type de donnée abstrait UNION-FIND :
↪ `type unionFind`
↪ `creerUF : int -> unionFind` : `creerUF n` crée une structure de n classes disjointes, les éléments sont les n entiers de 0 à $n-1$,

↪ `find : unionFind -> int -> int` qui donne la classe d'un élément i : deux éléments d'une même classe doivent avoir le même résultat
↪ `union : unionFind -> int -> int -> unit()` qui modifie une structure en joignant les classes correspondant à i et j.

Chaque classe d'équivalence peut être représentée par un arbre : chaque classe admet alors la racine comme représentant privilégié.
Par exemple pour la décomposition $\{0,4,5,8\}$, $\{1,3,7\}$, $\{2\}$, $\{6\}$ on peut avoir



Nous allons choisir une concrétisation par un tableau qui donne, pour chaque sommet, le numéro du père sauf pour les racines qui sont leur propre père.
L'exemple ci-dessus est codé par `[4; 3; 2; 3; 4; 8; 6; 3; 4]`.
Pour trouver le représentant de la classe on parcourt le tableau `t` jusqu'à avoir `t.(i) = i`.
Pour l'union de deux classes représentées par `i` et `j` on affecte `t.(j) <- i`.

```
let creerUF n =
  let t = Array.make n 0 in
  for i = 0 to (n-1) do
    t.(i) <- i done;
  t;;
```

qui peut se simplifier en

```
let creerUF n =
  Array.init n (fun i -> i);;
```

Les autres fonctions suivent les remarques

```
let rec find u i =
  if u.(i) <> i
  then find u (u.(i))
  else i;;
```

```
let union u i j =
  let k = find u i in
  let l = find u j in
  if k <> l
  then u.(k) <- l;;
```

On peut alors implémenter l'algorithme.

```
let kruskal g s0 =
  let n = taille g in
  let u = creerUF n in
  let t = creeTasVide (0,0,0) in
  let acm = ref [] in
  List.iter (fun a -> ajouterTas a t) (aretes g);
  while not (estTasVide t) do
    let (s1, s2, d) = extraireTas t in
    if (find u s1) <> (find u s2)
    then begin union u s1 s2;
               acm := (s1, s2) :: !acm end done;
 !acm;;
```

Code V.4 Algorithme de Kruskal

Preuve et complexité

L'algorithme de Kruskal fournit un arbre couvrant minimum.
Si on a utilisé un tas pour la file de priorité sa complexité est un $\mathcal{O}(m\ln(m))$ avec $m = |A|$.

La démonstration est similaire au cas de l'algorithme de Prim, voir l'exercice **10**.

4 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 $W(c') \leqslant W(c)$.
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 : $d(s,s'') \leqslant d(s,s') + d(s',s'')$.

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 on 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.

Ex. 8 **Arêtes traversantes** Prouver que toute arête d'un arbre couvrant est une arête traversante pour une coupure.

Ex. 9 **Compression dans Union-Find** On peut accélérer la complexité amortie des fonctions d'une structure Union-Find en comprimant les chemins, c'est-à-dire en affectant `t.(j) <- k` pour tous les élément vus lors de la recherche de `find u i`.
Modifier la fonction `find` selon cette remarque.
On admet qu'alors la complexité amortie des opération peut être considérée comme constante.

Ex. 10 **Preuve** Prouver que l'algorithme de Kruskal calcule un arbre minimal.
Et prouver sa complexité.

Ex. 11 **Autre écriture de l'algorithme de Kruskal** Écrire l'algorithme de Kruskal en remplaçant l'usage d'une file de priorité par le tri de toutes les arêtes.
Quelle est la complexité ?

Ex. 12 **Composantes connexes** Déterminer les composantes connexes d'un arbre non orienté à l'aide d'une structure union-find. Quelle est la complexité ?

Chapitre VI

Flots

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 Algorithme de Ford et Fulkerson

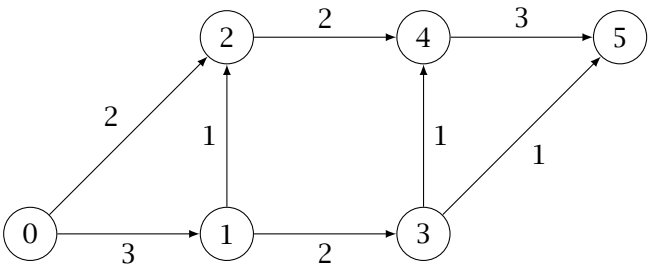
Dans cette sous-partie nous allons définir et caractériser la notion de flot maximal. Cette caractérisation va permettre d'imaginer une méthode de calcul qu'il faudra préciser ensuite.

1-1 Flot

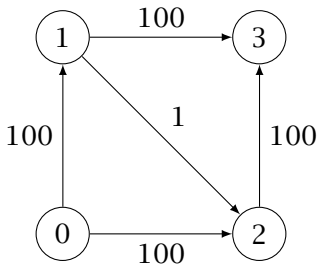
Réseau

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
↪ $\varphi(i,j) \leqslant c(i,j)$ 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, $\sum_{i \in T} \sum_{j \in T} \varphi(i,j) = 0$.

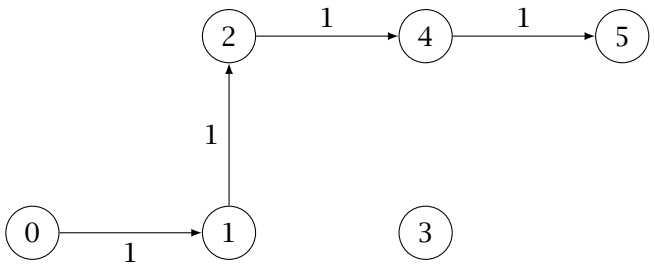
En déduire que $|\varphi| = \sum_{i \in S} \varphi(i,p)$.

Ex. 3 Flot associé à un chemin
On suppose que le graphe d'un réseau admet un chemin simple de s vers p :

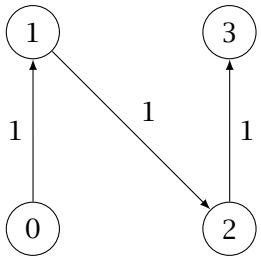
$$s = s_0 \to s_1 \to \cdots \to s_{r-1} \to s_r = p$$

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



$|\varphi_1| = 1$

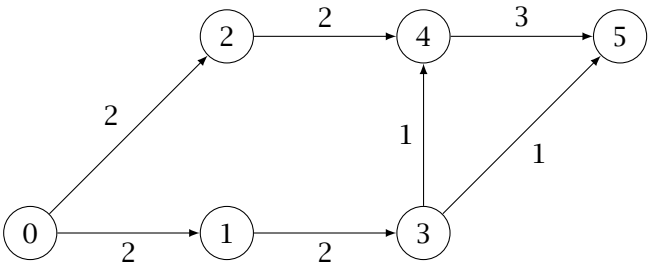


$|\varphi_2| = 1$

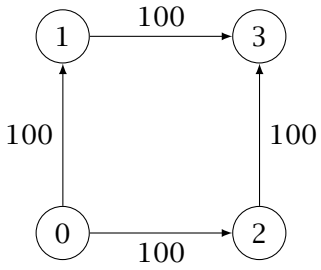
Propriétés

- Il existe au moins un flot : le flot nul défini par $\varphi(i, j) = 0$ pour tous sommets.
- Pour tout flot φ, on a $|\varphi| = \sum_{j \in S} \varphi(s, j) \leqslant \sum_{j \in S} c(s, j)$.
- 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



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



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

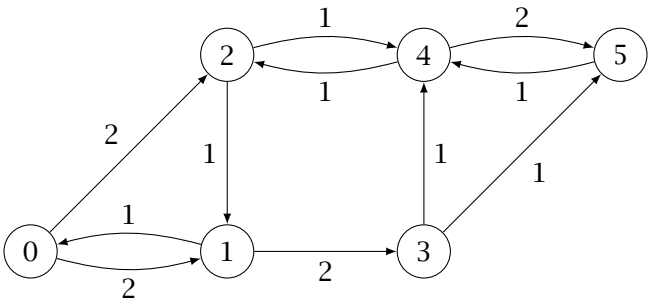
1-2 Chemin améliorant

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

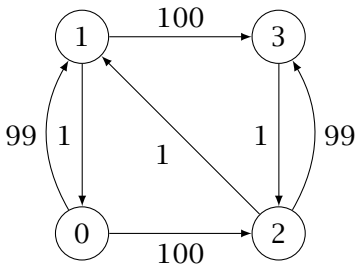
Réseau résiduel

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 **102**



R_{φ_1}



R_{φ_2}

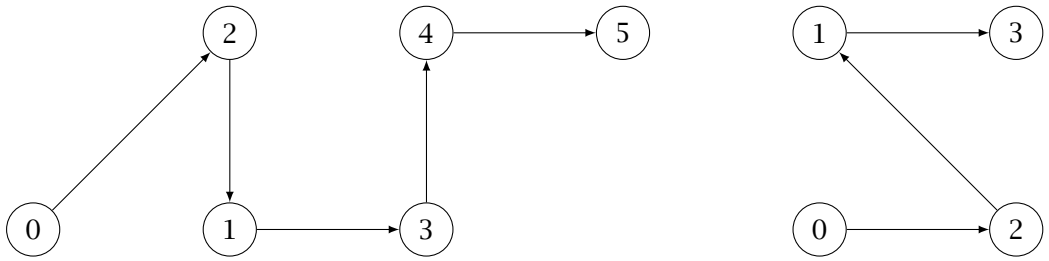
On remarque qu'on a enlevé des arêtes de R et introduit des arêtes supplémentaires en sens inverse par rapport à celles de R. Ces arêtes inversées correspondent à la possibilité de diminuer le flot qui suit l'arête originale.

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

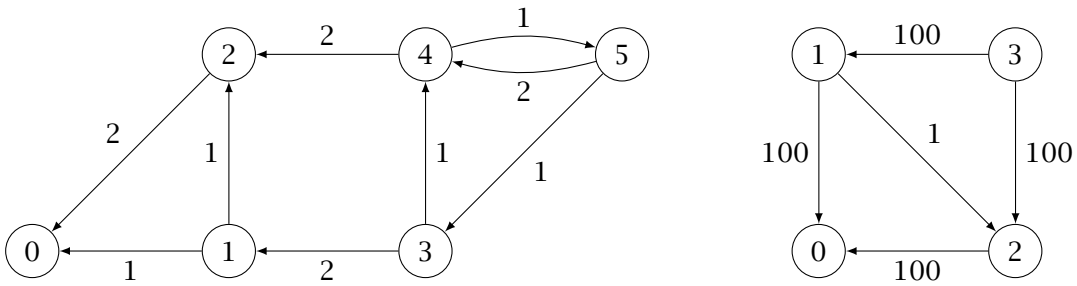
Si φ est un flot sur le réseau $R = (S, A, w, s, p)$, un chemin améliorant est un chemin de s vers p dans le réseau résiduel associé R_φ.

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.



$R_{\varphi_{1,M}}$ $R_{\varphi_{2,M}}$

L'algorithme 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.

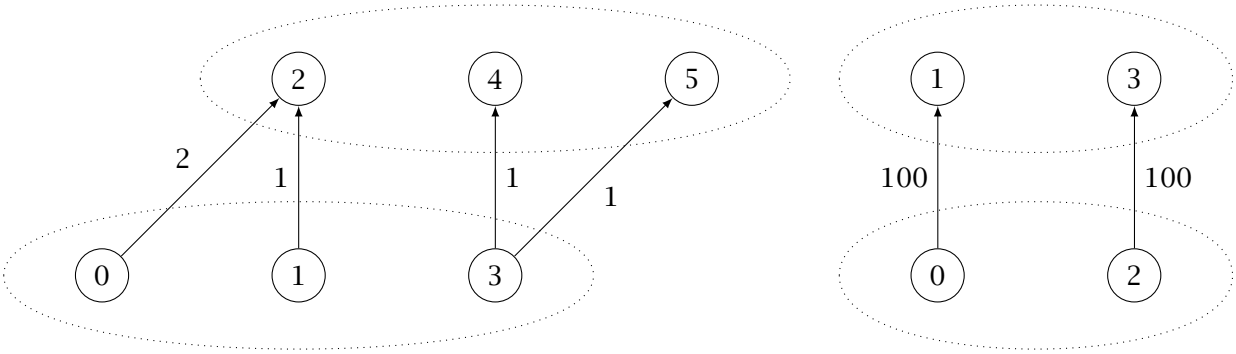
2 Coupure

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 $|\varphi| \leqslant c(S_s, S_p)$.
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 exercice 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 il faudrait s'assurer que la suite strictement croissante des valeurs des flots atteint sa limite après un nombre fini d'itérations. On peut construire des exemples où un choix du chemin améliorant conduit à un algorithme qui ne termine pas. La terminaison ne peut pas être prouvée dans ce cas.
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.

3-1 Implémentation

On choisit de coder les graphe 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. 10 **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 le l'arête (i,j) et l'ajouter au poids le l'arête (j,i) dans la matrice du graphe.

Ex. 11 **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. 12 **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.

3-2 Complexité

Dans cette partie on s'intéresse au comportement lors de l'usage d'un chemin augmentant.

Notations : rappels et définitions

↪ On par 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.
↪ On suppose que 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 $R_{\varphi'}$.
↪ Un arête de $R_{\varphi'}$ est dite **émergente** si elle n'appartient pas à R_φ.
↪ 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.
On utilisera de même $S_{\varphi'}$ et $d_{\varphi'}(t)$ pour $t \in S_{\varphi'}$.

Ex. 13 Caractérisations des arêtes critiques
Prouver que les arêtes critiques sont les arêtes de c de poids ω dans R_φ.

Ex. 14 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. 15 Décroissance de S_φ Prouver qu'on a $S_{\varphi'} \subset S_\varphi$.

Ex. 16 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 $d_\varphi(t) \leqslant d_{\varphi'}(t)$ pour tout $t \in S_{\varphi'}$.

Ex. 17 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 ?

Chapitre VII

Langages rationnels

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 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 $i+p-1$.

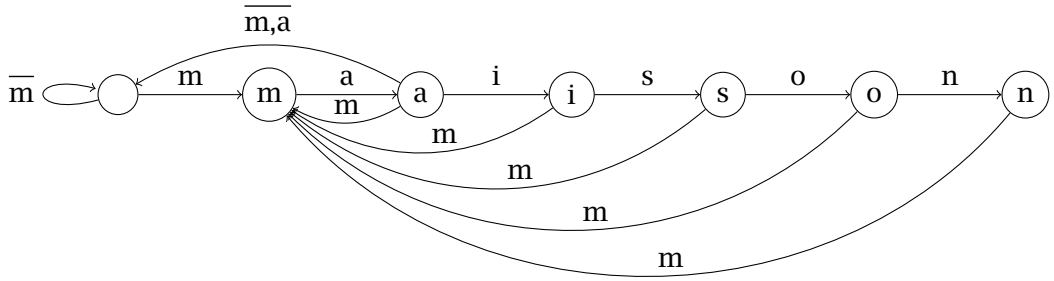
```
let rechercheTexte motif texte =
  let n = string_length texte in
  let p = string_length motif in
  let resultat = ref [] in
  for i = 0 to (n - p) do
    if (sub_string texte i p) = motif
    then result := i::(!resultat)
  done;
  !resultat;;
```

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`.

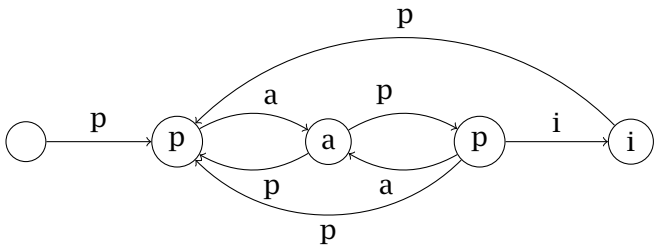
```
sub_string
string -> int -> int -> string
```

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 : $w = u_1 u_2 \cdots u_n$.

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 : $\overline{L} = \mathcal{A}^* \setminus L$.

11. Les **langages élémentaires** sont $\varnothing$, $\{\varepsilon\}$ 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 $w = u.v = w_1 \cdots w_{m+n}$ avec $w_i = u_i$ pour $1 \leqslant i \leqslant n$ et $w_i = v_{i-n}$ pour $n+1 \leqslant i \leqslant n+m$.

On prolonge la définition par $w.\varepsilon = \varepsilon.w = w$.
On remarque qu'on a, pour tous $u, v \in \mathscr{A}^*$, $|u.v| = |u| + |v|$.

Monoïde

$(\mathscr{A}^*, .)$ 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
↪ $\emptyset.L = L.\emptyset = \emptyset$.
↪ $\{\varepsilon\}.L = L.\{\varepsilon\} = L$.
↪ $\{u\}.\{v\} = \{u.v\}$.
↪ $\mathscr{A}^+ = \mathscr{A}.\mathscr{A}^*$.
↪ L'ensemble des mots de $\mathscr{A}$ qui commencent par a est $\{a\}.\mathscr{A}^*$.
↪ $\mathscr{A}^*.\{a\}.\mathscr{A}^*$ 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 $n \geqslant 1$.
↪ $L^1 = L$
↪ $L^n = \{w_1.w_2.\cdots.w_n ; w_i \in L \text{ pour tout } i\}$.
↪ $L^n.L^m = L^{n+m}$

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 $L = \mathscr{A}$ on retrouve bien l'ensemble de tous les mots : la notation $\mathscr{A}^*$ 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
$\mathscr{R}_0 = \{\varnothing, \{\varepsilon\}\} \cup \bigcup_{a \in \mathscr{A}} \{a\}$ et
$\mathscr{R}_{n+1} = \{L_1 \cup L_2 \; L_1, L_2 \in \mathscr{R}_n\} \cup \{L_1.L_2 \; L_1, L_2 \in \mathscr{R}_n\} \cup \{L^* \; L \in \mathscr{R}_n\}$
L'ensemble des langages rationnels est $\mathscr{R} = \bigcup_{n \in \mathbb{N}} \mathscr{R}_n$.

$\mathscr{R}$ contient tous les langages élémentaires
$\mathscr{R}$ est stable par union produit et étoile.

Démonstration
Pour tout $L \in \mathscr{R}_n$ on a $L = L \cup L \in \mathscr{R}_{n+1}$ donc $\mathscr{R}_n \subset \mathscr{R}_{n+1}$. Ainsi $\mathscr{R}$ contient $\mathscr{R}_0$ donc tous les langages élémentaires.
Si $L_1, L_2 \in \mathscr{R}$ alors il existe des entiers n et m tels que $L_1 \in \mathscr{R}_n$ et $L_2 \in \mathscr{R}_m$ donc $L_1, L_2 \in \mathscr{R}_p$ avec $p = \max(n, m)$.
On a alors $L_1 \cup L_2 \in \mathscr{R}_{p+1} \subset \mathscr{R}$, $L_1.L_2 \in \mathscr{R}_{p+1} \subset \mathscr{R}$ et $L_1^* \in \mathscr{R}_{p+1} \subset \mathscr{R}$.

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

Caractérisation

Si un ensemble $\mathscr{L}$ de langages (c'est-à-dire une partie de $\mathscr{P}(\mathscr{A}^*)$)
- contient tous les langages élémentaires
- est stable par union, produit et étoile

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

Démonstration

La démonstration se fait en montrant, par récurrence sur n,
que $\mathscr{R}_n$ est inclus dans $\mathscr{L}$ pour tout n.
1. *La première propriété implique $\mathscr{R}_0 \subset \mathscr{L}$.*
2. *Si on suppose qu'on a $\mathscr{R}_n \subset \mathscr{L}$ alors la stabilité de $\mathscr{L}$ par les opérations implique qu'on a $\{L_1 \cup L_2 \ L_1, L_2 \in \mathscr{R}_n\} \subset \mathscr{L}$, $\{L_1.L_2 \ L_1, L_2 \in \mathscr{R}_n\} \subset \mathscr{L}$ et $\{L^* \ L \in \mathscr{R}_n\} \subset \mathscr{L}$ d'où $\mathscr{R}_{n+1} \subset \mathscr{L}$*

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

On peut traduire cette propriété en disant que $\mathscr{R}$ 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
- $P(\varnothing)$, $P(\{\varepsilon\})$ et $P(\{a\})$ pour $a \in \mathscr{A}$ 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 $\mathscr{L}$ des langages L tels que $P(L)$ est vérifiée.

3 Expressions régulières

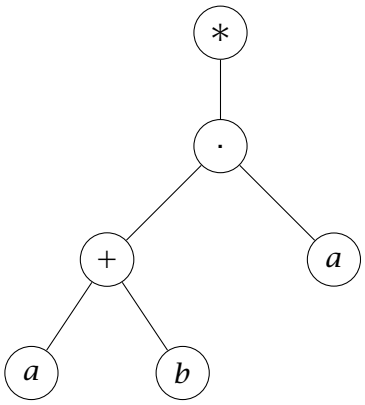
Dans cette partie nous allons donner, pour chaque langage rationnel, une construction à partir des langages élémentaires. L'alphabet $\mathscr{A}$ est fixé.

3-1 Arbre de construction

On considère les arbres dont les feuilles sont des langages élémentaires, on représentera le vide par **0** et ϵ par **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 $\mathscr{R}_n$ 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 $\Big((\{a\} \cup \{b\}).\{a\}\Big)^*$ 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).a)*)`.
On va donner une expression formelle de cette chaîne de caractère.

3-2 Langages réguliers

Expression régulière

$\mathscr{A}$ est un alphabet ne contenant pas les lettres "**0**" ,"**1**", "*", "+", ".", " "(, ")". Une expression régulière sur $\mathscr{A}$ est une expression de la forme
↪ **0**,
↪ **1**,
↪ `a` avec $a \in \mathscr{A}$,
↪ `(r1+r2)` avec `r1` et `r2` expressions régulières,
↪ `(r1.r2)` avec `r1` et `r2` 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 $\mathscr{A}$ on peut associer un langage par la construction récursive
↪ $L[\mathbf{0}] = \emptyset$,
↪ $L[\mathbf{1}] = \{\varepsilon\}$,
↪ $L[a] = \{a\}$ pour $a \in \mathscr{A}$,
↪ $L[(r1 + r2)] = L[r1] \cup L[r2]$,
↪ $L[(r_1.r_2)] = L[r1].L[r2]$,
↪ $L[(r*)] = (L[r])^*$.

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).a)*)` mais aussi par `((b.a+a.a)*)` ou `(((a.a)*).(((b.a).((a.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).a)*`, `(b.a+a.a)*` et `(a.a)*.(b.a.(a.a)*)*`.

Expressions équivalentes

Deux expressions régulières sont équivalentes si elles dénotent le même langage.

Parmi les langages élémentaires nous avons ajouté $\varnothing$ 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 **0**.

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 **0**.

1. *Pour les langages élémentaires la propriété est évidente.*

2. *On suppose que $P(L_1)$ et $P(L_2)$ sont vraies.*
   ↪ *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* `r1` *et* `r2` *ne contenant pas* **0**. *$L_1 \cup L_2$ est alors dénoté par* `(r1+r2)` *qui ne contient pas* **0**. *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* `r1` *et* `r2` *ne contenant pas* **0**. *L_1L_2 est alors dénoté par* `(r1.r2)` *qui ne contient pas* **0**. *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* **1** *donc $P(L^*)$ est vraie.*
↪ *Si L est non vide alors il est dénoté par une expression régulière* `r` *ne contenant pas* **0**. *L^* est alors dénoté par* `(r*)` *qui ne contient pas* **0**. *Ainsi $P(L^*)$ est vraie.*

Une formulation alternative est que toute expression régulière est équivalente à **0** ou à une expression régulière ne contenant pas **0**.

Une expression régulière est appelée réduite si elle ne contient ni **0** ni **1**.

Théorème : élimination du un

Tout langage rationnel est dénoté par une expression régulière de la forme **0**, **1**, `r` ou `(r+`**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 $\mathscr{A}$ est local
s'il existe $P \subset \mathscr{A}$, $S \subset \mathscr{A}$ et $F \subset \mathscr{A}^2$
tels que $u = u_1 u_2 \cdots u_n \in L \setminus \{\varepsilon\}$ est équivalent à
$u_1 \in P$, $u_n \in S$ et $u_i u_{i+1} \in F$ pour tout $i \in \{1, 2, \ldots, n-1\}$.

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

↪ $\varnothing$ et $\{\varepsilon\}$ sont locaux : ils sont définis par $P = F = S = \varnothing$.
↪ $L = \{a\}$ avec $a \in \mathscr{A}$ est local : $P = S = \{a\}$, $F = \varnothing$, $\varepsilon \notin L$.
↪ $L = \{(ab)^n \ ; \ n \in \mathbb{N}\}$ 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 $\mathscr{A} = \{a, b\}$, $L = \{a^n \ ; \ n \in \mathbb{N}\}$ 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 $N = \mathscr{A}^2 \setminus F$, la condition sur les facteurs devient $u_i u_{i+1} \notin N$.
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 structurelle 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 $\mathscr{A} = \{a\}$ 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 $v \prec u$. Prouver
↪ $v \prec u$ si et seulement si $|v| \leqslant |u|$ et $v_i = u_i$ pour $1 \leqslant i \leqslant |v|$.
↪ $\prec$ 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 $\mathscr{A}$ est un langage L sur $\mathscr{A}$ tel que l'égalité $x_1.x_2.\cdots.x_p = y_1.y_2.\cdots.y_q$ avec $x_1,\ldots, x_p, y_1,\ldots, y_q$ dans L entraîne $p = q$ et $x_i = y_i$ pour tout i.

1. Déterminer les codes parmi les langages finis suivants :
   - ↪ $L_1 = \{ab, baa, abba, aabaa\}$,
   - ↪ $L_2 = \{b, ab, baa, abaa, aaaa\}$,
   - ↪ $L_3 = \{aa, ab, aab, bba\}$,
   - ↪ $L_4 = \{a, ba, bba, baab\}$.
2. Soit $u \in \mathscr{A}^*$, montrer que $\{u\}$ est un code si et seulement si $u \neq \varepsilon$.
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 $\mathscr{A} = \{a, b\}$, 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 $\mathscr{A}$.
On suppose que a est une lettre de $\mathscr{A}$.
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 **0**, **1**, `r` ou `(r+1)` avec `r` expression rationnelle ne contenant ni **0** ni **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 $\mathscr{A} = \{a, b, c\}$ tels que deux lettres consécutives soient toujours distinctes.

Ex. 14 **Complémentaire** Donner une représentation régulière dénotant $\overline{L}$ 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

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 $(\mathscr{A}, S, \delta)$ où
1. $\mathscr{A}$ 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 $S \times \mathscr{A}$ vers S, la fonction de transition.

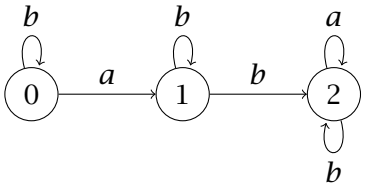
Une machine est caractérisée par la fonction de transition ; en effet l'alphabet et l'ensemble des états sont les composantes de son ensemble de départ. Voici un exemple.

	0	1	2
a	1	1	2
b	0	2	2

Transition

Une transition d'une machine $(\mathcal{A}, S, \delta)$ 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 $s.u = \big((\cdots((s.x_1).x_2).\cdots.x_{n-1}).x_n\big)$.
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 $(s, s_1, s_2, ..., s_n)$ 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 $Q = (\mathscr{A}, S, \delta, s_0, T)$ est l'ensemble $L(Q)$ de tous les mots reconnus par Q.

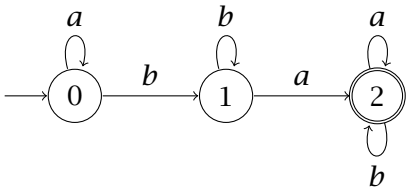
$$L(Q) = \{u \in \mathscr{A}^* \ ; \ s_0.u \in T\}$$

2. Un langage L est **reconnaissable** s'il existe un automate déterministe complet Q tel que $L = L(Q)$.

$Q = (\mathscr{A}, S, \delta, s_0, T)$ est un automate.
↪ Si $T = \emptyset$ alors $L(Q) = \emptyset$.
↪ Si $T = S$ alors $L(Q) = \mathscr{A}^*$.
↪ $L(Q)$ contient ε si et seulement si s_0 appartient à T.

Pour représenter graphiquement un automate
↪ l'état initial est indiqué par une flèche rentrante
↪ 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 $T = \{2\}$.

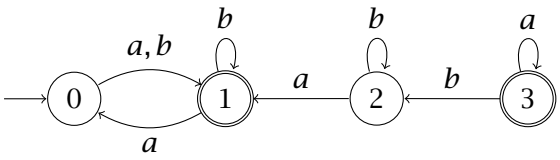


Le langage reconnu est l'ensemble des mots contenant ba.

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

$Q = (\mathscr{A}, S, \delta, s_0, T)$ est un automate déterministe complet.
Un état $s \in S$ est dit accessible s'il existe $u \in \mathscr{A}^*$ 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 $Q = (\mathscr{A}, S, \delta, s_0, T)$ alors, pour tout $s \in S'$ et pour tout $a \in \mathscr{A}$, $\delta(s, a)$ 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 $S' \times \mathscr{A}$ 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 $Q = (\mathscr{A}, S, \delta, s_0, T)$, l'automate émondé associé à Q est l'automate déterministe complet

$$Q' = (\mathscr{A}, S', \delta, s_0, T \cap S')$$

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 $\mathscr{A}$ est $\overline{L} = \mathscr{A}^* \setminus L$.

Complémentaire

Le complémentaire d'un langage reconnaissable est reconnaissable.

Démonstration : soit L reconnaissable. On considère un automate $Q = (\mathscr{A}, S, \delta, s_0, T)$ qui le reconnaît. On a a alors $u \in \overline{L}$ équivalent à $s_0.u \notin T$, c'est-à-dire à $s_0.u \in \overline{T} = S \setminus T$.
Ainsi $\overline{L}$ est reconnu par l'automate $(\mathscr{A}, S, s_0, \delta, \overline{T})$.

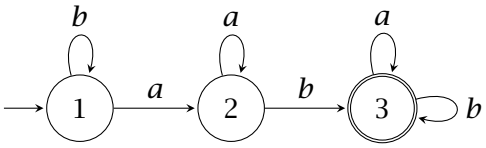
Pour l'union et le produit nous auront besoin d'une construction de machine.

Produit de machines

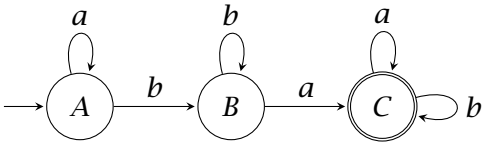
Si $M_1 = (\mathscr{A}, S_1, \delta_1)$ et $M_2 = (\mathscr{A}, S_2, \delta_2)$ sont deux machines sur le même alphabet leur produit $M_1 \odot M_2$ est la machine $(\mathscr{A}, S_1 \times S_2, \delta_1 \times \delta_2)$ avec $\delta_1 \times \delta_2((s,t),a) = (\delta_1(s,a), \delta_2(t,a))$.

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 $\mathscr{A}$ d'où, par récurrence, pour tout mot de $\mathscr{A}^*$, $(s,t).u = (s.u, t.u)$.

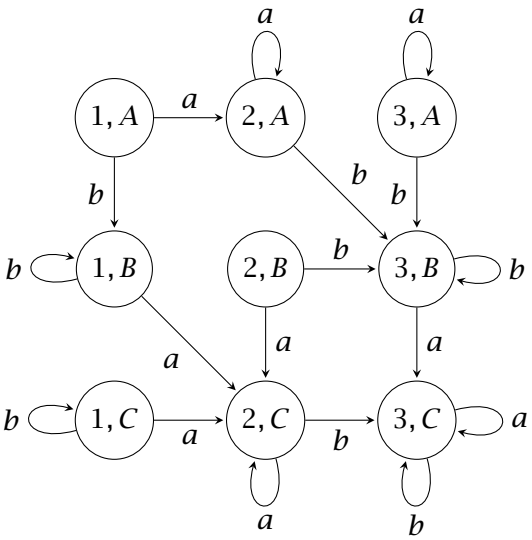
Exemple L'automate suivant reconnaît L_1, ensemble des mots contenant ab.



L'automate suivant reconnaît L_2, ensemble des mots contenant ba.

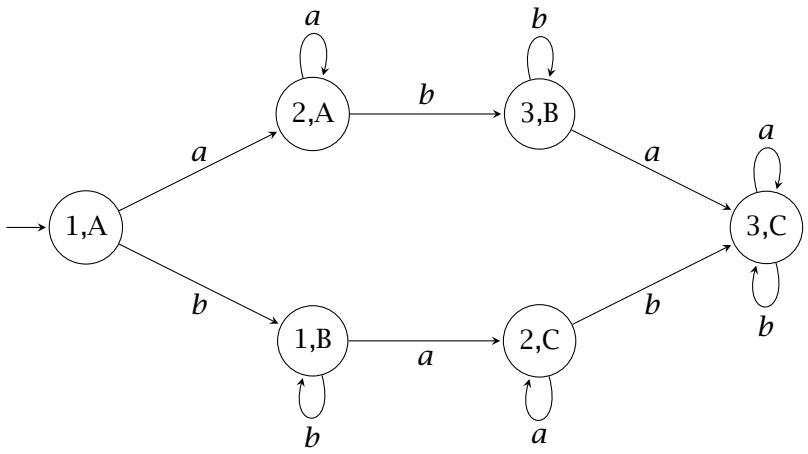


Le produit des machines associées est



Dans le cas de deux automates $Q_1 = (\mathscr{A}, S_1, \delta_1, s_{1,0}, T_1)$ et $Q_2 = (\mathscr{A}, S_2, \delta_2, s_{2,0}, T_2)$ 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 $(3,A)$.



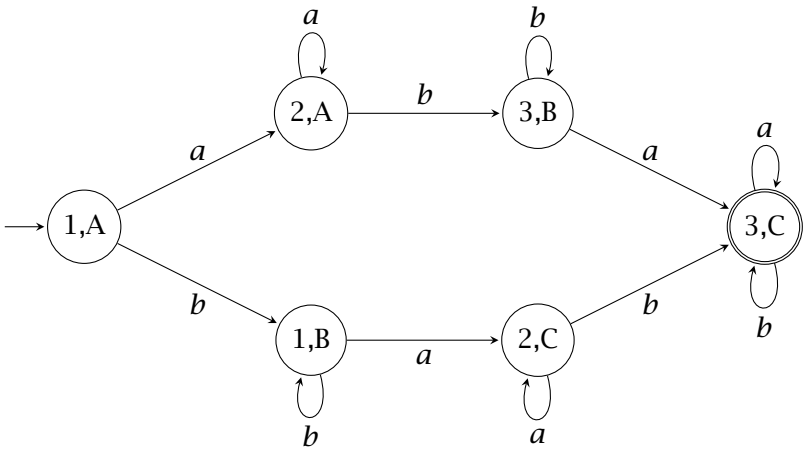
On remarque qu'un mot u est reconnu par Q_1 (resp. Q_2) si et seulement si $(s_{1,0},s_{2,0}).u \in T_1 \times S_2$ (resp. $(s_{1,0},s_{2,0}).u \in S_1 \times T_2$). Ainsi on a le théorème

Intersection et union

L'intersection de deux langages reconnaissables est reconnaissable.
L'union de deux langages reconnaissables est reconnaissable.

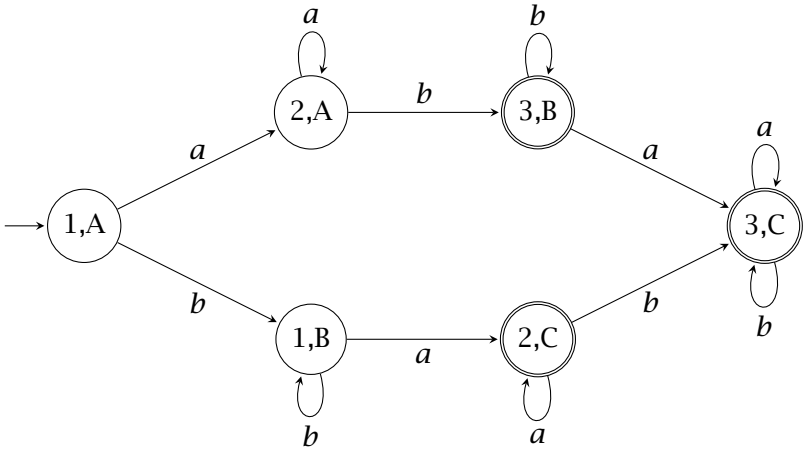
Démonstration
Si L_1 *est reconnu par* $(\mathcal{A}, S_1, \delta_1, s_{1,0}, T_1)$ *et* L_2 *par* $(\mathcal{A}, S_2, \delta_2, s_{2,0}, T_2)$ *alors*
$L_1 \cup L_2$ *est reconnu par* $(\mathcal{A}, S_1 \times S_2, \delta_1 \times \delta_2, (s_{1,0}, s_{2,0}), T_1 \times S_2 \cup S_1 \times T_2)$ *et*
$L_1 \cap L_2$ *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



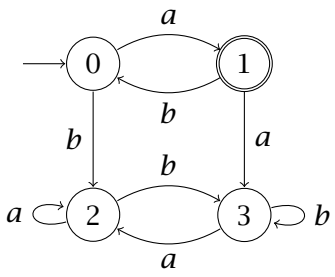
C'est aussi l'ensemble des mots commençant par $b^m a^n b$ ou $a^m b^n a$ avec $m \geqslant 1$ et $n \geqslant 1$.

$L_1 \cup L_2$ est reconnu par l'automate



1-5 Automates déterministes incomplets

Nous avons vus 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 $Q = (\mathscr{A}, S, \delta, s_0, T)$
↪ où $\mathscr{A}$ est un alphabet (fini),
↪ S est un ensemble fini d'états,
↪ δ est une application définie sur une **partie** de $S \times \mathscr{A}$ 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

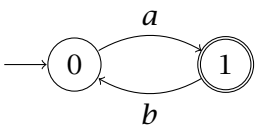
$Q = (\mathscr{A}, S, \delta, s_0, T)$ est un automate déterministe incomplet.
$u = u_1 u_2 \cdots u_p$ est un mot sur $\mathscr{A}$.
Si $s_0.u_1 = s_1$, $s_1.u_2 = s_2$, …, $s_{k-1}.u_k = s_k$ sont définis mais $\delta(s_k, u_{k+1})$ n'est pas défini (avec $k < p$) on dit que u est un **mot bloquant**.
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

$Q = (\mathscr{A}, S, \delta, s_0, T)$ est un automate déterministe.
Un état $s \in S$ est dit co-accessible s'il existe $u \in \mathscr{A}^*$ non bloquant tel que $s.u$ appartient à T.
Si S' est l'ensemble des états co-accessibles de Q alors
$Q' = (\mathscr{A}, S', \delta, s_0, T \cap S')$ 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.

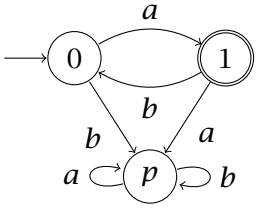
Complétion

$Q = (\mathscr{A}, S, \delta, s_0, T)$ 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 $S' \times \mathscr{A}$ 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
↪ $\delta'(p,x) = p$

$Q' = (\mathscr{A}, S', \delta', s_0, T)$ 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.

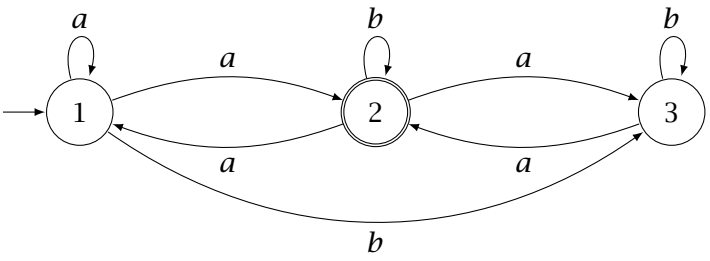
On peut remplacer la fonction de transition par son graphe $G_\Delta = \{(s,x,s') \ ; \ s' \in \Delta(s,x)\}$.
G_Δ est une partie de $S \times \mathcal{A} \times S$, c'est l'ensemble des transitions.
Δ est alors défini par $\Delta(s,x) = (\{s\} \times \{x\} \times S) \cap G_\Delta = \{s' \in S \ ; \ (s,x,s') \in G_\Delta\}$.

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 $s \xrightarrow{x_1} s_1 \xrightarrow{x_2} s_2 \xrightarrow{x_3} \cdots \xrightarrow{x_{n-1}} s_{n-1} \xrightarrow{x_n} s_n$.
  ↪ s est l'origine du calcul,
  ↪ s_n est l'extrémité du calcul,
  ↪ $u = x_1 x_2 \cdots x_n$ 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

$Q = (\mathscr{A}, S, \Delta, I, T)$ est un automate non-déterministe.

- L'automate des parties associé à Q est $Q' = (\mathscr{A}, \mathscr{P}(S), \delta, I, \mathscr{F})$ avec
  ↪ $\delta(E,x) = \bigcup_{s\in E} \Delta(s,x) = \{s' \in S \ ; \ \exists s \in E, \ (s,x,s') \in G_\Delta\}$.
  ↪ $\mathscr{F} = \{E \subset S \ ; \ E \cap T \neq \emptyset\}$
- Q' est un automate déterministe complet.
- $L(Q') = L(Q)$.

On dit aussi que Q' est le déterminisé de Q.

Démonstration

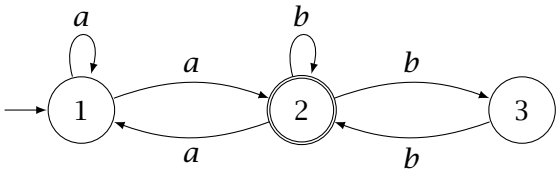
1. *I est une partie de S donc $I \in \mathscr{P}(S)$.*
   De même $\delta(E,x) \in \mathscr{P}(S)$ pour tout $E \in \mathscr{P}(S)$ et pour tout $x \in \mathscr{A}$.
   Q' est bien un automate déterministe complet.

2. *Si $u = x_1x_2\cdots x_n$ appartient à $L(Q)$ alors il existe un calcul réussi*
   $s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} s_2 \xrightarrow{x_3} \cdots \xrightarrow{x_{n-1}} s_{n-1} \xrightarrow{x_n} s_n$ *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' : $I \xrightarrow{x_1} E_1 \xrightarrow{x_2} E_2 \xrightarrow{x_3} \cdots \xrightarrow{x_{n-1}} E_{n-1} \xrightarrow{x_n} E_n$
   Or $s_n \in E_n$ et $s_n \in T$ donc $S_n \cap T \neq \emptyset$ d'où $S_n \in \mathscr{T}$.
   Ainsi $I.u \in \mathscr{T}$ dans Q' donc $u \in L(Q')$: on a prouvé $L(Q) \subset L(Q')$.

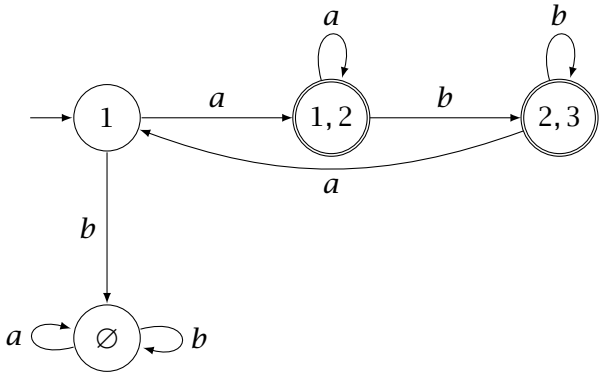
3. *Inversement si $u \in L(Q')$ alors son parcours est de la forme*

$I = E_0 \xrightarrow{x_1} E_1 \xrightarrow{x_2} \cdots \xrightarrow{x_{n-1}} E_{n-1} \xrightarrow{x_n} E_n$ avec $E_n \in \mathscr{T}$ donc il existe $s_n \in E_n \cap T$.
La transition $E_{n-1} \xrightarrow{x_n} E_n$ 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 $s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} \cdots \xrightarrow{x_{n-1}} s_{n-1} \xrightarrow{x_n} s_n$
$s_0 \in E_0 = I$, $s_n \in T$ 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'	$\{1\}$	$\{1,2\}$	$\{2,3\}$	$\varnothing$
a	$\{1,2\}$	$\{1,2\}$	$\{1\}$	$\varnothing$
b	$\varnothing$	$\{2,3\}$	$\{2,3\}$	$\varnothing$



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

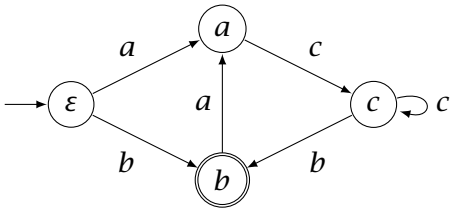
L est un langage local sur $\mathscr{A}$ défini par (P, F, S).

Automate de Glushkov

L'automate de Glushkov associé à L est l'automate incomplet $Q = (\mathscr{A}, \mathscr{A} \cup \{\varepsilon\}, \delta, \varepsilon, S')$ où δ est définie par
↪ $\delta(\varepsilon, x)$ existe si et seulement si $x \in P$, alors $\delta(\varepsilon, x) = x$,
↪ $\delta(x, y)$ 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 $\mathscr{A} = \{a, b, c\}$ défini par $P = \{a, b\}$, $S = \{b\}$, $F = \{ac, ba, cb, cc\}$ et ne contenant pas ε.

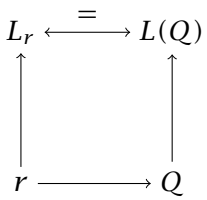


On remarquera que toutes les transitions de même étiquette ont la même extrémité.

Théorème de Glushkov

Si Q est l'automate de Glushkov associé à L alors Q reconnaît L.

On peut synthétiser ce résultat



3-2 Morphismes

On considère une fonction f d'un alphabet $\mathscr{B}$ vers un alphabet $\mathscr{A}$.

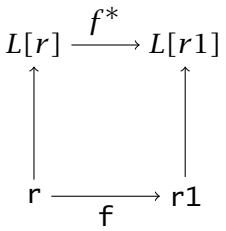
1. Les mots sont des produits de lettres : on peut donc définir une fonction f^* de $\mathscr{B}^*$ vers $\mathscr{A}^*$ par $f^*(\varepsilon) = \varepsilon$ (le premier ε est le mot vide de $\mathscr{B}^*$, le second celui de $\mathscr{A}^*$)
   et $f^*(u.x) = f^*(u).f(x)$ pour $u \in \mathscr{B}^*$ et $x \in \mathscr{B}$.
   On prolonge f^* sur les langages sans changer le nom :
   si L est un langage sur $\mathscr{B}$, $f^*(L) = \{f^*(u)\ ;\ u \in L\}$ est un langage sur $\mathscr{A}$.

2. De même on peut prolonger f de manière inductive sur l'ensemble des expressions régulières sur $\mathscr{B}$:
   `f(0) = 0`, `f(1) = 1`, `f(a) = b` avec $b = f(a)$,
   `f(r1 + r2) = (f(r1) + f(r'))`, `f(r1.r2) = (f(r1).f(r2))` et `f(r*) = (f(r))*`.

Transport des expressions régulières

Si f est une fonction de $\mathscr{B}$ vers $\mathscr{A}$ alors $L[f(r)] = f^*(L[r])$
pour toute expression régulière `r` sur $\mathscr{B}$.

On a donc un transport depuis les expressions régulières vers les langages.



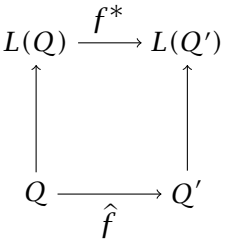
Dans ce diagramme les deux chemins de `r` à $L[r1]$ donnent le même résultat.

3. f peut servir aussi à transformer les automates (non déterministes) sur l'alphabet $\mathscr{B}$.
   Si $Q = (\mathscr{B}, S, \Delta, I, T)$ on note $\widehat{f}(Q) = (\mathscr{A}, S, \Delta', I, T)$ 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 $\mathscr{B}$ vers $\mathscr{A}$ alors
pour toute expression régulière r sur $\mathscr{B}$.

On a donc un transport depuis les automates vers les langages.



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.
   dénoté 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)`.

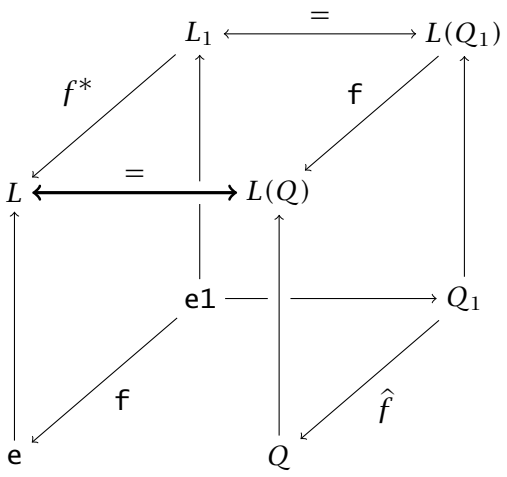
3. On note $L_1 = L[e1]$, L_1 est local car `e1` est linéaire.
4. On associe à L_1 son automate de Glushkov, Q_1.
5. On transforme Q_1 en Q en remplaçant les étiquettes à l'aide de f : $Q = \widehat{f}(Q_1)$.
6. On peut, si besoin, déterminiser Q.

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.

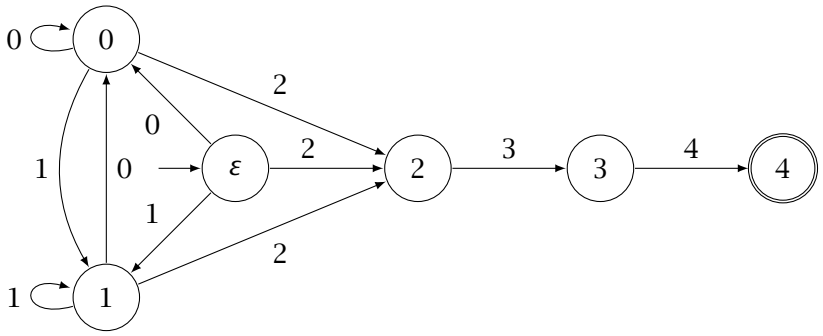
1. $L = L[e] = L[f(e1)] = f^*(L[e1]) = f^*(L_1)$ (transport des expressions régulières)
2. $L_1 = L(Q_1)$ (théorème de Glushkov)
3. $L(Q) = L(\widehat{f}(Q_1)) = f^*(L(Q_1))$ (transport des automates)
4. Ainsi $L = f^*(L_1) = f^*(L(Q_1)) = L(Q)$.



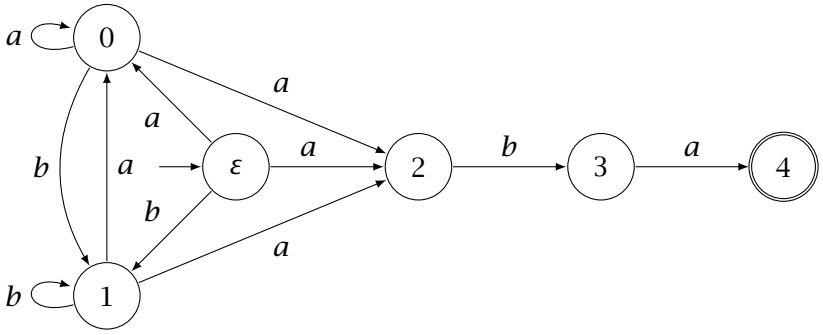
3-4 Un exemple

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]`.

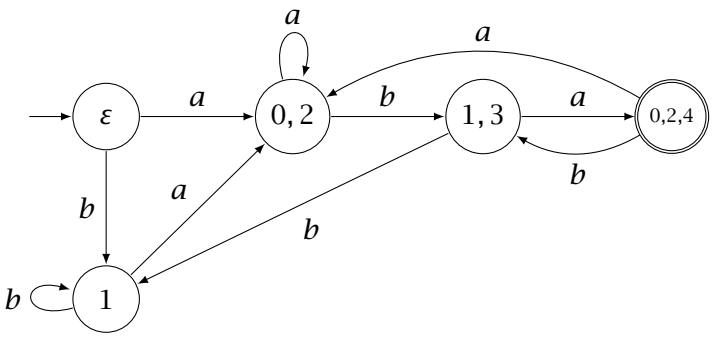
3. $L_1 = L[e1]$ ne content pas ε.
   On calcule $P = \{0, 1, 2\}$, $S = \{4\}$, $F = \{00, 01, 10, 11, 02, 12, 23, 34\}$.
4. L'automate Q_1 est calculé.



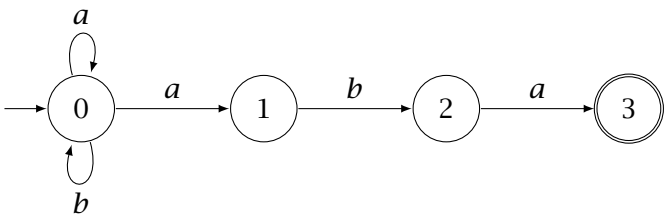
5. On traduit en l'automate Q, non déterministe.



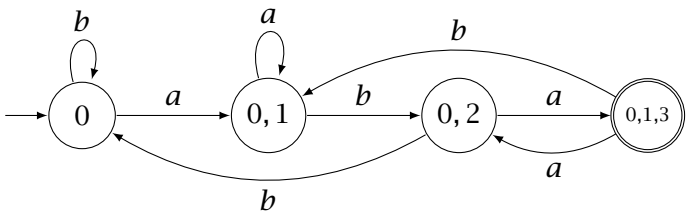
6. On peut le déterminiser.



On pouvait aussi trouver directement un automate non déterministe



puis le déterminiser



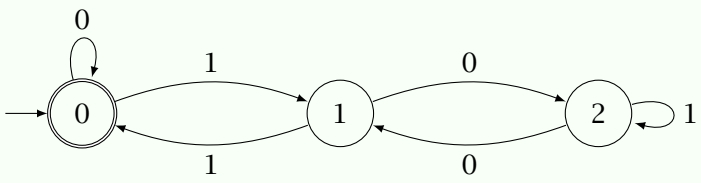
4 Exercices

4-1 Automates déterministes

Ex. 1 **Produit de 2 mots** Prouver que $s.(u.v) = (s.u).v$ pour $u, v \in \mathscr{A}^*$.

Ex. 2 **3 occurences** $\mathscr{A}$ désigne l'alphabet $\{a, b\}$ et L l'ensemble des mots qui contiennent au moins 3 occurrences de la lettre a : $L = \{u \in \Sigma^* / |u|_a \geqslant 3\}$.
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 $\mathscr{A} = \{a, b\}$ 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 $u = u_1 u_2 \cdots u_n$ le miroir de u est $u^T = u_n u_{n-1} \cdots u_1$.
Pour tout langage L le 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 $Q = (\mathscr{A}, S, \delta, s_0, T)$ est local si, pour tout $x \in \mathscr{A}$, $\delta(s, x)$ 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 $\mathscr{B}$ vers $\mathscr{A}$ alors $f^*(u.v) = f^*(u).f^*(v)$ pour tous mots u et v de $\mathscr{B}^*$.
Prouver que si L_1 et L_2 sont des langages sur $\mathscr{B}$ 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 $L[f(r)] = f^*(L[r])$.

Ex. 16 **Transport des automates** Prouver que $L(\widehat{f}(Q)) = f^*(L(Q))$.

Chapitre IX

Langages rationnels compléments

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

$Q = (\mathscr{A}, S, \Delta, I, T)$ 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 $(\mathscr{A}, S, \Delta, \{s\}, \{t\})$ 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 $(\mathscr{A}, S, \Delta, \{s\}, \{t\})$.

- 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 $Q = (\mathscr{A}, S, \Delta, I, T)$ est un automate tel que $S = \{0, 1, 2, \ldots, N-1\}$
$L^k(s,t)$ est l'ensemble des étiquettes des calculs entre s et t
ne passant que par des états de $\{0, 1, 2, \ldots, k-1\}$
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 $\{1,2,...,k\}$.

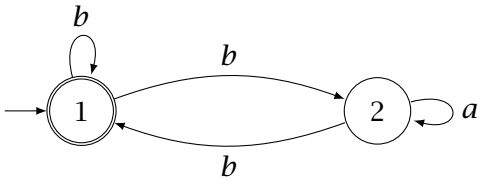
$$L^{k+1}(s,t) = L^k(s,t) \cup L^k(s,k).(L^k(k,k))^*.L^k(k,t)$$

Comme les langages $L^k(s,t)$ sont finis donc rationnels et que l'on n'effectue que des opération 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 : un 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 ?

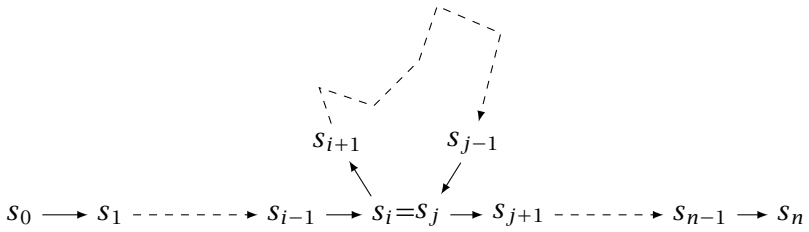
2 Lemme de l'étoile (ou de pompage)

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) $Q = (\mathscr{A}, S, \Delta, I, T)$.
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 $n \geqslant N$ on considère un calcul réussi de u dans Q :

$$s_0 \xrightarrow{x_1} s_1 \xrightarrow{x_2} \cdots \xrightarrow{x_{N-1}} s_{N-1} \xrightarrow{x_N} s_N \xrightarrow{x_{N+1}} \cdots \xrightarrow{x_{n-1}} s_{n-1} \xrightarrow{x_N} s_n$$

L'ensemble de $N+1$ éléments $\{s_0, s_1, \ldots, s_N\}$ est inclus dans S de cardinal N donc il existe deux indices i et j distincts tels que $0 \leqslant i < j \leqslant N$ et $s_i = s_j$.



Si on définit $u_1 = x_1 \cdots x_i$, $u_2 = x_{i+1} \cdots x_j$ et $u_3 = x_{j+1} \cdots x_n$ on a
↪ $u = u_1.u_2.u_3$,
↪ $|u_1 u_2| = j \leqslant N$,
↪ $|u_2| = j - i \geqslant 1$,
↪ $s_0.u_1 = s_i$
↪ $s_i.u_2 = (s_0.u_1).u_2 = s_0.(u_1.u_2) = s_j = s_i$.
↪ $s_i.u_3 = s_j.u_3 = (s_0.(u_1.u_2)).u_3 = s_0.u = s_n \in T$.

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 $u = u_1u_2u_3$ avec $|u_1u_2| \leqslant N$, $|u_2| \geqslant 1$ et $\forall k \in \mathbb{N}$, $u_1u_2^ku_3 \in L$.

Exemples

- Soit $L = \{a^nb^n \ ; \ n \in \mathbb{N}\}$.
  Si L était reconnaissable on pourrait, pour l'entier N du lemme de l'étoile, décomposer $u = a^Nb^N \in L$ en $u_1.u_2.u_3$ avec $|u_1.u_2| \leqslant N$ et $|u_2| \geqslant 1$.
  On en déduit que $u_1 = a^p$, $u_2 = a^q$ avec $q \geqslant 1$ et $u_3 = a^rb^N$ avec $p + q + r = N$.
  On devrait avoir $u_1u_2^ku_3 = a^{N+(k-1)q}b^N \in L$ 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 $\begin{cases} |u|_a = |u|_b \\ \text{si } u = v.w, \text{ alors } |v|_a \geqslant |v|_b \end{cases}$
  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 $u = a^Nb^N \in D$ en $u_1.u_2.u_3$ avec $|u_1.u_2| \leqslant N$ et $|u_2| \geqslant 1$.
  On en déduit que $u_1 = a^p$, $u_2 = a^q$ avec $q \geqslant 1$ et $u_3 = a^rb^N$ 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 $k \neq 1$: 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 $\mathscr{A}$ et si u est un mot ($u \in \mathscr{A}^*$) le langage dérivé (à gauche) de L par u est $u^{-1}.L = \{v \in \mathscr{A}^* \ / \ u.v \in L\}$.
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. $u^{-1}.\mathscr{A}^* = A^*$.
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 $\mathscr{A} = \{a, b\}$ 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 $\mathscr{A}^*$.

Ex. 4 **Un autre exemple**
On pose $L = \{a^n b^n \ ; \ n \in \mathbb{N}\}$ sur $\mathscr{A} = \{a, b\}$.
Prouver que $(a^p)^{-1}.L = \{a^n b^{n+p} \ ; \ n \in \mathbb{N}\}$ pour $p \geqslant 1$.
Prouver que $(a^p.b^q)^{-1}.L = \{b^{p-q}\}$ pour $p \geqslant q \geqslant 1$.

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
↪ Λ est l'ensemble des résiduels de L, il contient $L = \varepsilon^{-1}.L$,
↪ Λ_T est l'ensemble des résiduels contenant ε,
↪ $\delta_L(\lambda, x) = x^{-1}.\lambda$ 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 $\varepsilon \in u^{-1} \cdot L$. 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 $\Lambda = \{u^{-1}.L \ ; \ u \in \mathscr{A}^*\}$.

Soit $Q = (\mathscr{A}, S, \delta, s_0, T)$ un automate déterministe émondé complet qui reconnaît L.
On note $L(s)$ le langage reconnu par $(\mathscr{A}, S, \delta, s, T)$ pour tout sommet de Q.
Si s est un sommet de Q, il est accessible donc il existe $u \in \mathscr{A}^*$ 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 reconaissant 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 faire[4].

On se donne un automate **déterministe** $Q = (\mathscr{A}, S, \delta, s_0, F)$ 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.
- Pour chaque état s on note $Q_t = (\mathscr{A}, S, \delta, s, T)$ 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 $x \in \mathscr{A}$.

↪ 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.

[4] On peut se référer au document assez exhaustif suivant :
http://www.di.ens.fr/ jv/HomePage/dea/MinimizeAutomata.pdf

↪ On note $\widehat{S}$ l'ensemble des classe d'équivalences.
↪ $\widehat{T}$ est l'ensemble des classes d'équivalences qui sont incluses dans T. On a vu que $\sigma(s) \in \widehat{T}$ 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 $\widehat{\delta}$ de $\widehat{S} \times \mathscr{A}$ vers $\widehat{S}$ par $\widehat{\delta}(\sigma(s), x) = \sigma(s.x)$

Ex. 8 Automate réduit
Montrer que $\widehat{Q} = (\mathscr{A}, \widehat{S}, \widehat{\delta}, \sigma_0, \widehat{T})$ 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 $L^{(n)} = \{u \in L \ ; \ |u| \leqslant n\}$, 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 $[s \equiv_{k+1} t] \iff [s \equiv_k t \text{ et } \forall x \in \mathscr{A}\ s.x \equiv_k t.x]$.

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 $\equiv$.

Liste des programmes

Liste des définitions