	OCaml	Python
test d'égalité	`=`	`==`
test de différence	`<>`	`!=`
division euclidienne	`/`	`//`
modulo	`mod`	%
et	`&&`	`and`
ou	`||`	`or`

Opérateurs en OCaml/Python

	OCaml
Définition	`let r = ref ...`
Accéder à la valeur	`!r`
Modifier la valeur	`r := ...`

Références

- `()` est une valeur de type `unit`, qui signifie rien.
- Si `f : 'a -> 'b -> 'c` alors `f` prend deux arguments de type `'a` et `'b` et renvoie un résultat de type `'c`.
  De plus, `f x` (application partielle) est la fonction de type `'b -> 'c` qui à `y` associe `f x y`.
- On créé souvent une liste avec une fonction récursive :

```
let rec range n = (* renvoie [n - 1; ...; 1; 0] *)
    if n = 0 then []
    else (n-1)::range (n - 1)
```

On peut éventuellement utiliser une référence sur une liste :

```
let range n =
    let l = ref [] in (* moins idiomatique *)
    for i = n - 1 downto 0 do
        l := i::!l
    done; !l
```

- **Impossible de modifier une liste** `l`.
  `e::l` renvoie une **nouvelle liste** mais ne modifie pas `l`.
  `let l = [] in ...` ne sert à rien.
- **Impossible d'écrire** `l.(i)` pour une liste `l` : il faut utiliser une fonction récursive pour parcourir une liste.

```
let rec appartient e l = match l with
    | [] -> false
    | t::q -> e = t || appartient e q
```

- Chaque cas d'un `match` sert à **définir de nouvelles variables**, et ne permet pas de comparer des valeurs.

Exemple : Dans `appartient`, il ne faut pas écrire `| e::q -> ...` (ceci remplacerait le `e` en argument).
Pour tester l'égalité : `| t::q -> if t = e then ...` ou `| t::q when t = e -> ...`.

- Les indices d'un tableau à n éléments vont de 0 à $n-1$:

```
let appartient e t =
    let r = ref false in
    for i = 0 to  n - 1 do
        if t.(i) = e then r := true
    done; !r
```

- **Impossible de renvoyer une valeur à l'intérieur d'une boucle** (pas de `return` en OCaml) :

```
let appartient e t =
    for i = 0 to n - 1 do
        if t.(i) = e then true (* NE MARCHE PAS !!! *)
    done; false
```

- Types union et enregistrement :

```
(* type union : l'un OU l'autre *)
type 'a option = None | Some of 'a
(* on utilise match (et fonction récursive)
   sur les types union *)
let add x y = match x, y with
    | None, _ | _, None -> None
    | Some x1, Some y1 -> Some (x1 + y1)

(* type enregistrement : l'un ET l'autre *)
type fraction = { num : int; denum : int }
let f = { num = 1; denum = 4 }
f.denum (* donne 4 *)
```

Remarque : l'égalité (avec `=`) est automatiquement définie avec un nouveau type. Exemple : si `t1` et `t2` sont des arbres binaires, alors `t1 = t2` vaut `true` si `t1 = V` et `t2 = V` ou si `t1 = N(r1, g1, d1)`, `t2 = N(r2, g2, d2)` avec `r1 = r2`, `g1 = g2` et `d1 = d2`.

- Quelques fonctions utiles (avec les équivalents sur `Array`) :
  - `Array.make_matrix n p e` renvoie une matrice $n \times p$ dont chaque élément est `e`
  - `List.iter f l` appelle `f` sur chaque élément de `l`
  - `List.map f [a1; ...; an]` renvoie `[f a1; ...; f an]`
  - `List.filter f l` renvoie la liste des `e` tels que `f e`
  - `List.exists f l` et `List.for_all f l`.

Type	Exemple	Obtenir valeur	Modification	Taille	Création
`array`	`[|1; 2|] : int array`	`t.(i)`	`t.(i) <- ...`	`Array.length` (en O(1))	`Array.make n e` (en O(n))
`string`	`"abc" : string`	`s.[i]`		`String.length` (en O(1))	`String.make n e`
`list`	`[1; 2] : int list`	Fonction récursive		`List.length` (en O(n))	
`tuple`	`(1, "abc", [1; 2]) : int*string*int list`	`let a, b, c = t`			

Remarque : une « liste » Python est en réalité un tableau.

- Un arbre binaire (enraciné) est défini par :

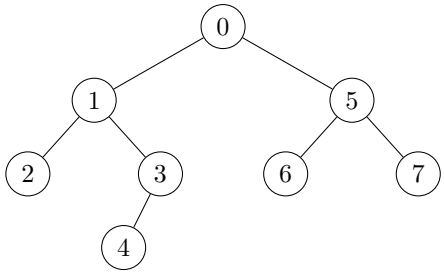
```
type 'a arbre = V | N of 'a * 'a arbre * 'a arbre
```

  Si `a = N(r, g, d)`, `r` est la **racine** (ou étiquette) de `a`, `g` est son **sous-arbre gauche** et `d` est son **sous-arbre droit**.

- Un arbre binaire est **strict** si chaque noeud a exactement 0 ou 2 fils. On peut alors utiliser un type différent, si les étiquettes sont sur les feuilles seulement :

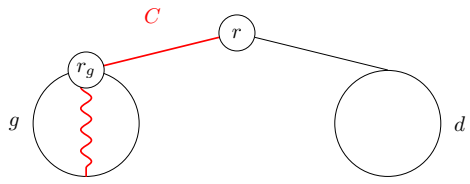
```
type 'a arb_strict =
    F of 'a | N of 'a arb_strict * 'a arb_strict
```

- Une **feuille** est un noeud sans fils.
  La **profondeur** d'un noeud est la longueur du chemin (en nombre d'arêtes) depuis la racine jusqu'à ce noeud.
  La **hauteur** $h(a)$ d'un arbre est la profondeur maximum d'une feuille. Par convention, on choisit souvent $h(V) = -1$.



Arbre binaire de hauteur 3, racine 0, feuilles $2, 4, 6, 7$

- Soit $a = N(r, g, d)$ un arbre binaire.
  Alors $\boxed{h_a = 1 + \max(h(g), h(d))}$.
  Preuve : Soit C un chemin de longueur maximum de la racine de a à une feuille.
  C passe soit dans g, soit dans d (et pas dans les deux). Supposons que C passe dans g, l'autre cas étant symétrique.



  Soit C_g la partie de C qui est incluse dans g.
  Supposons que C_g ne soit pas un chemin de longueur maximum de la racine à une feuille dans g. Il existe alors un chemin C'_g plus long que C_g dans g. Mais alors la concaténation de l'arête de r à r_g (racine de g) et du C'_q est plus long que C, ce qui est une contradiction.
  Donc C_g est un plus long chemin de r_g à une feuille de g : sa longueur est donc $h(g)$ par définition. D'où $h_a = h(g) + 1$.
  Si C passe par d, on a de même : $h_a = h(d) + 1$.
  Comme la hauteur est le maximum sur tous les chemins possibles : $h_a = 1 + \max(h(g), h(d))$.

```
let rec h a = match a with
    | V -> -1
    | N(_, g, d) -> 1 + max (h g) (h d)
```

  Complexité : linéaire en est le nombre de noeuds $n(a)$ de a, car on `h a` effectue un appel récursif sur chaque noeud de a.

- Exercice : Le **diamètre** $diam(a)$ d'un arbre a est la longueur maximum d'un chemin entre 2 feuilles de a.
  Si $a = N(r, g, d)$, montrer que $diam(a) = \max(diam(g), diam(d), h(g) + h(d) + 2)$ et en déduire une fonction pour calculer $diam(a)$.
  Solution : Il y a 3 possibilités pour un chemin C dans a : soit C passe par r, soit C est entièrement dans g ou dans d.
  La longueur maximum d'un chemin dans g est $diam(g)$.
  La longueur maximum d'un chemin dans d est $diam(d)$.
  Soit C un chemin de a, passant par r_a et de longueur $l(C)$ maximum. Montrons que $l(C) = h(g) + h(d) + 2$. Pour cela, notons C_g la partie de C dans g. Alors C_g est un chemin maximum de g depuis la racine de g (si ce n'était pas le cas, on pourrait trouver un chemin C'_g plus long et remplacer C_g par C'_g dans C pour obtenir une contradiction). Donc $l(C_g) = h(g)$, par définition de la hauteur. De même pour $l(C_d)$.
  Donc $l(C) = l(C_g) + l(C_d) + 2 = h(g) + h(d) + 2$ (le $+2$ venant des 2 arêtes issues de r_a). Comme d_a correspond à la longueur du chemin maximum parmi ces 3 possibilités : $diam(a) = \max(diam(g), diam(d), h(g) + h(d) + 2)$

```
let rec h t = function (* hauteur *)
  | V -> -1
  | N(_, g, d) -> 1 + max (h g) (h d);;

let rec diam t = function
  | V -> 0
  | N(_, g, d) -> max (max (diam g) (diam d)) (h g + h d + 2)
```

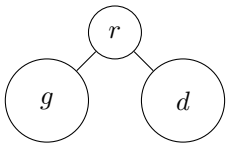
  Si `t` a n noeuds, `diam t` appelle n fois `h` donc est en $O(n^2)$.
  On peut obtenir une complexité linéaire en calculant diamètre et hauteur simultanément :

```
let rec diam t = function (* renvoie diamètre, hauteur *)
  | V -> -1, -1
  | N(_, g, d) ->
    let dg, hg = diam g in
    let dd, hd = diam d in
    max (max dg dd) (hg + hd + 2), 1 + max hg hd
```

- Le nombre n de noeuds et la hauteur h d'un arbre binaire a vérifie : $\boxed{h + 1 \leq n \leq 2^{h+1} - 1}$.
  Preuve : Un chemin de longueur h a $h + 1$ sommets donc $n \geq h + 1$.
  On montre par récurrence qu'il y a au plus 2^p sommets à profondeur p, d'où $n \leq \sum_{p=0}^{h} 2^p = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1$.

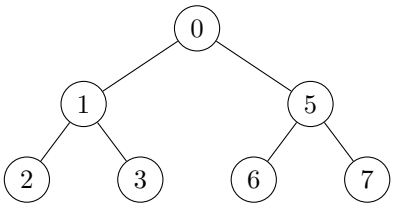
- Exercice : Soit $f(a)$ le nombre de feuilles et $n(a)$ le nombre de noeuds d'un arbre binaire strict a. Alors $n(a) = 2f(a) - 1$.

Preuve : Récurrence sur le nombre de noeuds (ou : sur la hauteur...).
Soit $P(n) : n = 2f(a) - 1$ pour tout arbre binaire strict a à $n \geq 1$ noeuds.
$P(1)$ est vrai car un arbre à 1 noeud possède 1 feuille.
Soit $n \geq 2$. Supposons $P(k)$ pour tout $k < n$.
Soit $a = N(r, g, d)$ un arbre à $n \geq 2$ noeuds.



Par récurrence sur g et d, on a $n(g) = 2f(g) - 1$ et $n(d) = 2f(d) - 1$.
Donc $n(a) = n(g) + n(d) + 1 = 2f(g) - 1 + 2f(d) - 1 + 1 = 2(f(g) + f(d)) - 1 = 2f(a) - 1$.
Remarque : Si on note $n_i(a)$ le nombre de noeuds **internes** de a (c'est-à-dire les noeuds qui ne sont pas des feuilles), on a $n(a) = n_i(a) + f(a)$ et donc $f(a) = n_i(a) + 1$.

- Un arbre binaire est **complet** si tous les niveaux sont remplis : il y a 2^p noeuds à profondeur p.
  Le nombre de noeuds d'un arbre binaire complet de hauteur h est donc $\sum_{p=0}^{h} 2^p = 2^{h+1} - 1$.



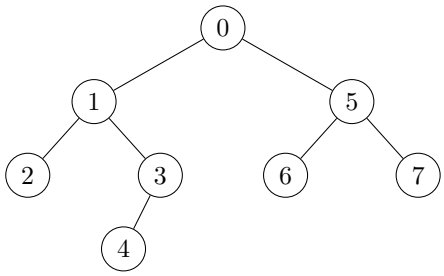
Arbre binaire complet

- a est complet $\iff$ $a = V$ où $a = N(r, g, d)$ avec g, d complets et de même hauteur.
  On en déduit une fonction récursive pour savoir si un arbre est complet :

```
let rec complet = function
    | V -> true
    | N(r, g, d) ->
        complet g && complet d && h g = h d
```

On peut obtenir une complexité linéaire avec la même astuce que pour `diam`.

- On distingue 3 types de parcours en profondeur sur un arbre binaire $N(r, g, d)$:
  - **Parcours préfixe** : visite r, puis g (appel récursif), puis d (appel récursif).
  - **Parcours infixe** : visite g (appel récursif), puis r, puis d (appel récursif).
  - **Parcours postfixe** : visite g (appel récursif), puis d (appel récursif), puis r.



Parcours préfixe $0, 1, 2, 3, 4, 5, 6, 7$
Infixe $2, 1, 4, 3, 0, 6, 5, 7$
Suffixe $2, 4, 3, 1, 6, 7, 5, 0$

Implémentation naïve en $O(n^2)$ où n = nombre de noeuds :

```
let rec prefix = function
    | V -> []
    | N(r, g, d) -> r::(prefix g)@(prefix d)

let rec infixe = function
    | V -> []
    | N(r, g, d) -> (infixe g)@r::(infixe d)
```

Ou en $O(n)$ avec accumulateur :

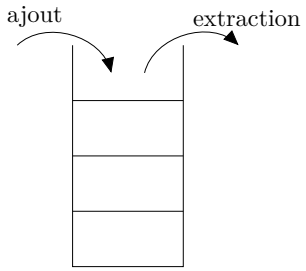
```
let rec prefix a =
    let rec aux acc = function
        | V -> acc
        | N(r, g, d) -> r::(aux (aux acc g) d)
    in aux [] a

let rec infixe a =
    let rec aux acc = function
        | V -> acc
        | N(r, g, d) -> aux (r::aux acc d) g
```

- Une structure de données **abstraite** est la description des opérations que doit comporter cette structure (exemple : dictionnaire).
  Une **implémentation** d'une structure de données est la réalisation pratique de cette structure. Il peut y avoir plusieurs implémentations de la même structure de données abstraite (exemple : un dictionnaire peut être implémenté par un arbre binaire ou une table de hachage).

- Une structure de données est **mutable** si on peut modifier ses éléments. Sinon, elle est **immuable** (ou : persistante).
  Par exemple, un algorithme de tri sera de type `'a list -> 'a list` pour une liste (on ne peut pas modifier la liste en argument donc on en renvoie une nouvelle) et `'a array -> unit` pour un tableau (la modification d'un tableau à l'intérieur de la fonction se répercute à l'extérieur).
  - Mutable : `array`, `ref`, `mutable` dans un type enregistrement.
  - Immuable : `list`, `string`, `tuple`, `arbre`.

  Les structures immuables sont plus adaptées à la programmation récursive et les structures mutables à la programmation impérative (boucles, références).

- Une **pile** est une structure de donnée possédant trois opérations :
  - `push` : Ajout d'un élément au dessus de la pile.
  - `pop` : Extraction (suppression et renvoi) de l'élément au dessus de la pile. Ainsi, c'est toujours le dernier élément rajouté qui est extrait.
  - `is_empty` : Test pour savoir si la pile est vide.

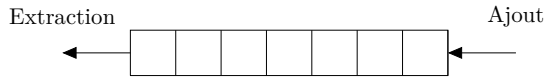


  C'est une structure LIFO : Last In First Out (dernier entré, premier sorti).
  On peut implémenter une pile avec une liste, où la tête de liste est le dessus de la pile.

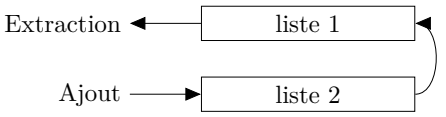
```
let push x s = x::s

(* pop s renvoie (x,s') où x est l'élément extrait
et s' le reste de la pile *)
let pop s = match s with
    | [] -> failwith "pop"
    | x::s -> (x,s)

let is_empty s = s = []
```

  Applications :
  - Pile d'appel qui stocke les appels de fonctions en cours d'exécution.
  - Passer de récursif à itératif en simulant des appels récursifs (ex : DFS).

- Une **file** est une structure de donnée possédant trois opérations:
  - Ajout d'un élément à la fin de la file.
  - Extraction (suppression et renvoi) de l'élément au début de file. Ainsi, l'élément extrait est l'élément le plus ancien.
  - Test pour savoir si la file est vide.



  C'est une structure FIFO : First In First Out (premier entré, premier sorti).
  Implémenter une file avec une liste OCaml ne serait pas efficace, car accéder au dernier élément est en $O(n)$.
  À la place, on peut implémenter une file avec deux listes `f1` et `f2`, en remplaçant `f1` par `f2` (inversé) si `f1` est vide :

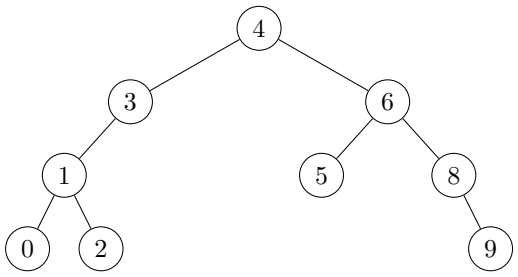



```
type 'a file = 'a list * 'a list

let empty = ([], []) (* file vide *)

let is_empty (f1,f2) = f1 = [] && f2 = []

let add x (f1, f2) = (f1, x::f2)

(* suppose que la file n'est pas vide *)
let rec extract (f1, f2) = match f1 with
    | [] -> extract ([], List.rev f2)
    | x::q1 -> (x, (q1, f2))
```

  Dans le pire des cas, un appel à `extract` sera en $O(n)$ car `List.rev` est en $O(n)$...
  Par contre, la complexité amortie (c'la complexité moyennée sur plusieurs appels) est meilleure. En effet, si on effectue n ajouts et n extractions dans un ordre quelconque (en partant d'une file vide), chaque élément est « renversé » exactement une fois, donc la complexité totale des `List.rev` est $O(n)$, d'où une complexité amortie de $O(1)$.
  Remarque : On peut aussi donner une implémentation mutable d'une file avec une tableau.
  Application : Parcours en largeur.

- Un **dictionnaire** associe à chaque **clé** une **valeur** avec les opérations :
  - Ajouter une association (clé, valeur)
  - Supprimer une association (clé, valeur)
  - Recherche de la valeur associée à une clé

- On peut implémenter un dictionnaire par arbre binaire de recherche.
  Un **arbre binaire de recherche** (ABR) est un arbre binaire tel que, pour chaque noeud d'étiquette r et de sous-arbres g et d, r est supérieur à toutes les étiquettes de g et inférieur à toutes les étiquettes de d.
  Il faut donc une relation d'ordre sur les étiquettes.



Exemple d'ABR

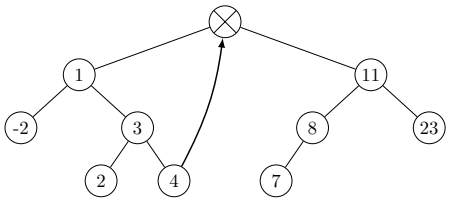
Pour chercher si un élément e appartient à un ABR $N(r, g, d)$, il suffit de regarder dans g si $e < r$ ou dans d si $e > r$.

```
let rec appartient e a = match a with
    | V -> false
    | Noeud (r, g, d) ->
        if e = r then true
        else if e < r then appartient e g
        else appartient e d
```

`appartient` est en $O(h)$ où h est la hauteur de l'arbre, car on ne parcourt qu'une branche de l'arbre.
Pour ajouter un élément e à un ABR $N(r, g, d)$, on peut soit le placer dans g si $e < r$ soit dans d si $e > r$, de façon à conserver la propriété d'ABR.

```
let rec ajoute e = function
    | V -> N(e, V, V) (* ajoute e comme feuille *)
    | N(r, g, d) when e < r -> N(r, ajoute e g, d)
    | N(r, g, d) -> N(r, g, ajoute e d)
```

On peut supprimer un élément en le remplaçant par le maximum de son sous-arbre gauche (qui est tout à droite de l'arbre), de façon à conserver la propriété d'ABR.




```
(* supprimer_max a renvoie le maximum de l'arbre
et l'arbre sans ce maximum *)
let rec supprimer_max = function
    | V -> min_int, V
    | N(r, g, V) -> r, g
    | N(r, g, d) -> let m, d' = del_max d in
                    m, N(r, g, d')

(* supprimer e a renvoie un ABR contenant les
mêmes éléments que a sauf e *)
let rec supprimer e = function
    | E -> E
    | N(r, g, d) when e = r -> let m, g' = del_max g in
                               N(m, g', d)
    | N(r, g, d) when e < r -> N(r, del e g, d)
    | N(r, g, d) -> N(r, g, del e d)
```

où chaque noeud contient un couple (clé, valeur) et les sous-arbres gauche et droit sont les sous-dictionnaires associés aux clés inférieures et supérieures à la clé du noeud.

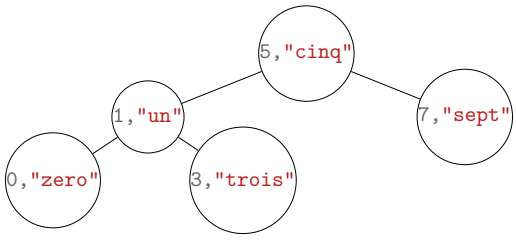
- On peut implémenter un dictionnaire avec un ABR dont les noeuds contiennent des couples (clé, valeur) où les étiquettes sont comparées sur les clés :

```
type ('k, 'v) dict =
    V | N of ('k * 'v) * ('k, 'v) dict * ('k, 'v) dict
(* 'k est le type des clés et 'v le type des valeurs *)
```

Les fonctions d'ajout et de suppression sont alors données par `ajoute`, `supprime`, et on peut adapter `appartient` pour pouvoir chercher une clé :

```
(* recherche k d renvoie None si k n'est pas une clé une d,
et Some v si k est associée à la valeur v *)
let rec recherche k = function
    | V -> None
    | N((k', v), g, d) when k = k' -> Some v
    | N((k', v), g, d) when k < k' -> recherche k g
    | N((k', v), g, d) -> recherche k d
```

Remarque : On a utilisé le type prédéfini `type 'a option = None | Some of 'a`.



Exemple de dictionnaire implémenté par ABR

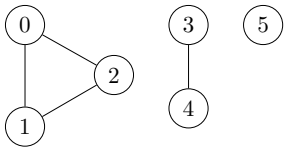
- Implémenter un dictionnaire par ABR demande une relation d'ordre sur les clés et toutes les fonctions ci-dessus sont en $O(h)$. Une autre possibilité est d'utiliser une table de hachage où les complexités peuvent être $O(1)$ en moyenne, mais il faut bien choisir la fonction de hachage.

Définitions

- Un **graphe non orienté** est un couple $G = (V, E)$ où V est un ensemble fini de **sommets** et E est un ensemble d'**arêtes**, chaque arête étant un ensemble de deux sommets de V.
  Si $e = \{u, v\} \in E$, on dit que u et v sont **adjacents** ou **voisins** et que ce sont les extrémités de e.
  Le **degré** $\deg(v)$ d'un sommet v est son nombre de voisins.
  Une **feuille** est un sommet de degré 1.
- Un **graphe orienté** est la même chose qu'un graphe non orienté, sauf que chaque arête est un couple (u, v) (représenté par $u \longrightarrow v$) au lieu d'un ensemble.
- Si G est un graphe non orienté à n sommets et p arêtes alors
$$p \leq \binom{n}{2} = O(n^2)$$
En effet, il y a $\binom{n}{2}$ arêtes possibles (il faut choisir les deux extrémités pour avoir une arête). De plus, un graphe avec toutes les arêtes possibles ($p = \binom{n}{2}$) est dit **complet**.
- (HP) Si $G = (V, E)$ est un graphe non orienté alors :
$$\sum_{v \in V} \deg(v) = 2|E|$$
Preuve : Récurrence sur le nombre d'arêtes ou double comptage ($\sum_{v \in V} \deg(v)$ compte deux fois chaque arête, puisqu'une arête a deux extrémités).
- (HP) Un graphe possède un nombre pair de sommets de degrés impairs.
  Preuve :
$$\underbrace{\sum_{\deg(v) \text{ pair}} \deg(v)}_{\text{pair}} + \sum_{\deg(v) \text{ impair}} \deg(v) = \underbrace{2|E|}_{\text{pair}}$$
- Un **chemin** est une suite d'arêtes consécutives différentes.
  La **longueur** d'un chemin est son nombre d'arêtes.
  La **distance** $d(u, v)$ de u à v est la plus petite longueur d'un chemin de u à v (∞ si il n'y a pas de chemin) : c'est une distance au sens mathématique, qui vérifie en particulier l'inégalité triangulaire.
  Dans le cas d'un graphe pondéré (avec des poids sur les arêtes), on somme les poids des arêtes du chemin.

Représentations

- Exemple de graphe avec ses représentations en OCaml :



Matrice d'adjacence

```
[|[|0; 1; 1; 0; 0; 0|];
[|1; 0; 1; 0; 0; 0|];
[|1; 1; 0; 0; 0; 0|];
[|0; 0; 0; 0; 1; 0|];
[|0; 0; 0; 1; 0; 0|];
[|0; 0; 0; 0; 0; 0|]|]
```

Liste d'adjacence

```
[|[1; 2];
[0; 2];
[0; 1];
[4];
[3];
[]|]
```

Une matrice adjacence d'un graphe non orienté est toujours symétrique.
Dans le cas d'un graphe pondéré : on met le poids de l'arête au lieu de 1 pour une matrice d'adjacence et, en général, un couple (voisin, poids) pour une liste d'adjacence.

- Pour un graphe orienté à n sommets et p arêtes :

	Matrice d'adjacence	Liste d'adjacence
complexité mémoire	$O(n^2)$	$\boxed{O(n+p)}$
tester si $\{u, v\} \in E$	$\boxed{O(1)}$	$O(\deg(u))$
parcourir voisins de u	$O(n)$	$\boxed{O(\deg(u))}$

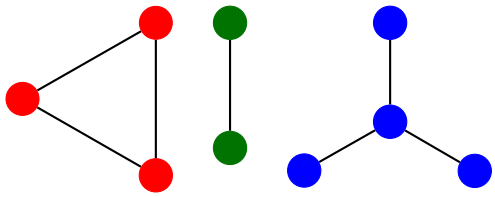
On utilise plutôt une liste d'adjacence pour un graphe avec peu d'arêtes et une matrice d'adjacence sinon.

Connexité, cycle

- Un graphe G non orienté est **connexe** si, pour tout sommets u et v, il existe un chemin entre u et v dans G.
- Un graphe G orienté est **fortement connexe** si, pour tout sommets u et v, il existe un chemin de u à v dans G.
- (HP) Un graphe connexe à n sommets a au moins $n - 1$ arêtes.
  Preuve : Soit $\mathcal{H}(n)$: « un graphe connexe à n sommets a au moins $n - 1$ arêtes ».
  $\mathcal{H}(1)$ est vraie car un graphe à 1 sommet possède 0 arête.
  Supposons $\mathcal{H}(n)$. Soit $G = (V, E)$ un graphe connexe à $n+1$ sommets.
  - Si G a un sommet v de degré 1 alors $G - v$ (G auquel on a enlevé le sommet v) est un graphe *connexe* à n sommets donc, par $\mathcal{H}(n)$, $G - v$ a au moins $n - 1$ arêtes. Donc G a au moins n arêtes.
  - Sinon, tous les sommets de G sont de degré ≥ 2.
    Alors $2|E| = \sum_{v \in V} \deg(v) \geq 2(n + 1) \geq 2n$.
    Donc $|E| \geq n$, ce qui montre $\mathcal{H}(n + 1)$.
- La relation suivante est une relation d'équivalence sur un graphe non orienté :
$$u \sim v \iff \text{il existe un chemin entre } u \text{ et } v$$

Les classes d'équivalences pour $\sim$ sont les sous-graphes connexes maximaux (au sens de $\subseteq$) de G, ils sont appelés **composantes connexes**.

Exemple : un graphe avec 3 composantes connexes.



Remarque : Un graphe est connexe ssi il contient une unique composante connexe.

- (HP) Un graphe acyclique contient un sommet de degré au plus 1.
  Preuve : Supposons que tous les sommets soient de degrés ≥ 2.
  Faisons partir un chemin depuis un sommet quelconque en visitant à chaque étape le sommet adjacent différent du prédécesseur (possible car les degrés sont ≥ 2). Comme le nombre de sommets est fini, ce chemin revient nécessairement sur un sommet déjà visité, ce qui donne un cycle.

- (HP) Un graphe acyclique à n sommets possède au plus $n-1$ arêtes.
  Preuve : Montrons par récurrence $\mathcal{H}(n)$: « un graphe acyclique à n sommets a au plus $n-1$ arêtes ».
  $\mathcal{H}(1)$ est vraie car un graphe à 1 sommet possède 0 arête.
  Supposons $\mathcal{H}(n)$. D'après le lemme, un graphe G acyclique à $n+1$ sommets possède un sommet v de degré ≤ 1.
  Comme G est acyclique, $G-v$ l'est aussi et a au plus $n-1$ arêtes, par $\mathcal{H}(n)$.
  Donc G a au plus $n-1+\deg(v) \leq n$ arêtes, ce qui montre $\mathcal{H}(n+1)$.

- Un graphe T à n sommets et p arêtes est un **arbre** s'il vérifie l'une des conditions équivalentes :
  1. T est connexe acyclique (définition).
  2. T est connexe et $p = n-1$.
  3. T est acyclique et $p = n-1$.

  Remarque : les arbres vus en 1ère année étaient enracinés. Ici il n'y a pas de racine.
  Preuve :
  $1 \implies 2$: Si T est connexe et acyclique alors $p \geq n-1$ (connexe) et $p \leq n-1$ (acyclique) donc $p = n-1$.
  $2 \implies 3$: Supposons T connexe et $p = n-1$.
  Par l'absurde, supposons que T contienne un cycle C. Soit e une arête de C.
  Montrons alors que $T-e$ est connexe.
  Soient donc u et v deux sommets de T. Comme T est connexe, il existe un chemin C' dans T entre u et v.
  Si C' ne contient pas e, C' est un chemin de u à v dans T.
  Si C' contient e, remplaçons e par le reste de C. C' est alors un chemin entre u et v.
  $T-e$ est donc connexe, ce qui contredit le fait que $T-e$ contienne n sommets et $n-2$ arêtes.
  $3 \implies 1$: Supposons T acyclique et $p = n-1$.
  Supposons par l'absurde que T ne soit pas connexe.
  Alors T a $k \geq 2$ composantes connexes T_1, ..., T_k. Notons n_i et p_i le nombre de sommets et d'arêtes de T_i.
  Comme T_i est connexe, $p_i \geq n_i - 1$. Donc $p = \sum p_i \geq \sum (n_i - 1) = n - k$, ce qui est absurde avec l'hypothèse $p = n-1$.

Parcours de graphe

- **Parcours en profondeur** : on visite les sommets le plus profondément possible avant de revenir en arrière.

```
let dfs (g : int list array) r =
    let n = Array.length g in
    let visited = Array.make n false in
    let rec aux v =
        if not visited.(v) then (
            visited.(v) <- true;
            (* traiter v *)
            List.iter aux g.(v)
        ) in
    aux r
```

Complexité avec représentation par liste d'adjacence :
Soit n le nombre de sommets et p le nombre d'arêtes.
`Array.make n false` est en $O(n)$.
Chaque arête est parcourue au plus une fois, d'où $O(p)$ appels récursifs de `aux`.
Au total : $\boxed{O(n+p)}$.
Remarque : La complexité est $O(n^2)$ avec une matrice d'adjacence.

- Application : recherche d'un cycle dans un graphe non orienté, en faisant attention à ne pas considérer une arête comme cycle (pas d'appel récursif sur le père).

```
let has_cycle (g : int list array) =
  let n = Array.length g in
  let visited = Array.make n false in
  let ans = ref false in
  let rec aux p u = (* p a permis de découvrir u *)
    if not visited.(u) then (
      visited.(u) <- true;
      List.filter ((<>) p) g.(u)
      |> List.iter (aux u)
    )
    else ans := true in
  for i = 0 to n - 1 do
    if not visited.(i) then aux i i
  done;
  !ans
```

- **Parcours en largeur** : on visite les sommets par distance croissante depuis le sommet de départ. Pour cela, on stocke les prochains sommets à visiter dans une file `q` (en utilisant ici le module `Queue` d'OCaml) :

```
let bfs (g : int list array) r =
    let seen = Array.make (Array.length g) false in
    let q = Queue.create () in
    let add v =
        if not seen.(v) then (
            seen.(v) <- true; Queue.add v q
        ) in
    add r;
    while not (Queue.is_empty q) do
        let v = Queue.pop q in
        (* traiter v *)
        List.iter add g.(v)
    done
```

- Application : calcul de distance (en nombre d'arêtes), en stockant des couples (sommet, distance) dans `q`.

```
let bfs g r =
    let dist = Array.make (Array.length g) (-1) in
    let q = Queue.create () in
    let add d v =
        if dist.(v) = -1 then (
            dist.(v) <- d;
            Queue.add (v, d) q
        ) in
    add 0 r;
    while not (Queue.is_empty q) do
        let u, d = Queue.pop q in
        List.iter (add (d + 1)) g.(u)
    done;
    dist (* dist.(u) est la distance de r à u *)
```

- Pour obtenir les plus courts chemins, on peut stocker le sommet `pred.(v)` qui a permis de découvrir `v` :

```
let bfs g r =
    let pred = Array.make (Array.length g) (-1) in
    let q = Queue.create () in
    let add p v = (* p est le père de v *)
        if pred.(v) = -1 then (pred.(v) <- p; Queue.add v q) :
    add r r;
    while not (Queue.is_empty q) do
        let v = Queue.pop q in
        List.iter (add v) g.(v)
    done;
    pred (* pred.(v) = père de v dans le parcours *)
```

On peut alors obtenir un plus court chemin de `r` à `v` en remontant les pères de `v` jusqu'à obtenir `r` :

```
let rec path pred v =
    if pred.(v) = v then [v]
    else v::path pred pred.(v)
```

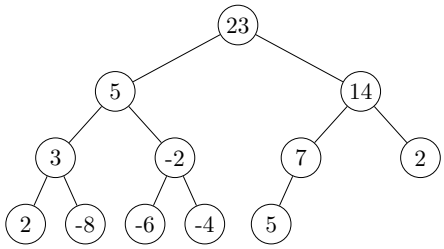
File de priorité

- Une **file de priorité max** (FP max) est une structure de données possédant les opérations suivantes :
  - Extraire maximum : supprime et renvoie le maximum
  - Ajouter élément
  - Tester si la FP est vide

  On définit une FP min en remplaçant maximum par minimum.
- Il est possible d'utiliser un arbre binaire de recherche pour implémenter une FP max, en utilisant le fait que le maximum est tout à droite de l'arbre. Les opérations d'ajout et d'extraction sont alors linéaires en la hauteur de l'arbre. On peut mettre à jour un élément (changer sa valeur) en le supprimant puis en le réajoutant (avec la nouvelle valeur).

Tas

- Un **tas max** est un arbre binaire presque complet (tous les niveaux, sauf le dernier, sont complets) où chaque noeud est plus grand que ses fils.
  Remarque : la racine est le maximum du tas.



Exemple de tas max

  Un tas min est comme un tas max, sauf que chaque noeud doit être plus petit que ses fils.
- Soit T un arbre binaire presque complet de hauteur h et avec n noeuds. Alors $h = \lfloor \log_2(n) \rfloor$ (donc $\boxed{h = \mathrm{O}(\log(n))}$).

  > Preuve : T contient plus de sommets qu'un arbre binaire complet de hauteur $h-1$ et moins de sommets qu'un arbre binaire complet de hauteur h, donc :
  >
  > $$\begin{aligned} & 2^h - 1 < n \leq 2^{h+1} - 1 \\ \implies & 2^h \leq n < 2^{h+1} \\ \implies & h \leq \log_2(n) < h+1 \\ \implies & h = \lfloor \log_2(n) \rfloor \end{aligned}$$

- On stocke les noeuds du tas dans un tableau `t` tel que :
  - `t.(0)` est la racine de `t`.
  - `t.(i)` a pour fils `t.(2*i + 1)` et `t.(2*i + 2)`, si ceux-ci sont définis.

  Le père de `t.(j)` est donc `t.((j - 1)/2)` (si $j \neq 0$).
  Exemple : le tas en exemple ci-dessus est représenté par `t = [|23; 5; 14; 3; -2; 7; 2; 2; -8; -6; -4; 5|]`.
- En pratique, comme on ne peut pas ajouter d'élément à un tableau, on utilise un tableau `t` plus grand que le nombre d'éléments du tas pour pouvoir y ajouter des éléments. On stocke le nombre d'éléments dans une variable `n`.

```
type 'a tas = {t : 'a array; mutable n : int}

let swap tas i j = (* échange tas.t.(i) et tas.t.(j) *)
    let tmp = tas.t.(i) in
    tas.t.(i) <- tas.t.(j);
    tas.t.(j) <- tmp
```

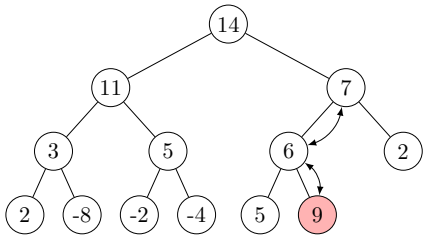
- On utilise deux fonctions auxiliaires pour rétablir la propriété de tas après modification :
  - `up tas i` : suppose que `tas` est un tas max sauf `tas.t.(i)` qui peut être supérieur à son père. Fait monter (on dit aussi percoler) `tas.t.(i)` de façon à obtenir un tas max.
  - `down tas i` : suppose que `tas` est un tas max sauf `tas.t.(i)` qui peut être inférieur à un fils. Fait descendre `tas.t.(i)` de façon à obtenir un tas max.

```
let rec up h i =
    let p = (i - 1)/2 in
    if i <> 0 && tas.t.(p) < tas.t.(i) then (
        swap h i p;
        up h p
    )

let rec down tas i =
  let m = ref i in (* maximum parmi i et ses fils *)
  if 2*i + 1 < tas.n && tas.t.(2*i + 1) > tas.t.(!m)
  then m := 2*i + 1;
  if 2*i + 2 < tas.n && tas.t.(2*i + 2) > tas.t.(!m)
  then m := 2*i + 2;
  if !m <> i then (
    swap h i !m;
    down h !m
  )
```

- Pour ajouter un élément à un tas : l'ajouter comme dernière feuille (dernier indice du tableau) puis faire remonter.
  Complexité : $\mathrm{O}(h) = \boxed{\mathrm{O}(\log(n))}$.

```
let ajoute tas e =
    tas.t.(tas.n) <- e;
    up h tas.n;
    tas.n <- tas.n + 1
```

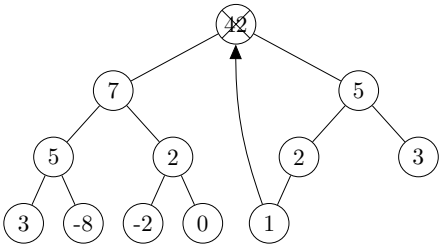
  Exemple : ajout de 9 dans un tas.



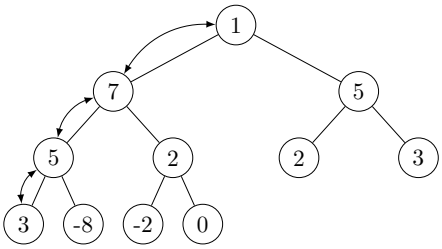
- Pour extraire le maximum d'un tas : échanger la racine avec la dernière feuille puis descendre la nouvelle racine.
  Complexité : $\mathrm{O}(h) = \boxed{\mathrm{O}(\log(n))}$.

```
let rec extract_max h =
    swap h 0 (h.n - 1);
    h.n <- h.n - 1;
    down h 0;
    h.a.(h.n)
```

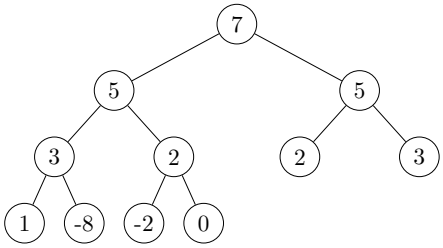
Exemple :



Suppression de la racine qu'on remplace par la dernière feuille



Percolation



Tas obtenu après extraction du maximum

Applications

- **Tri avec une file de priorité** : on ajoute tous les éléments dans un tas puis on les extrait un par un.
  On obtient ainsi le **tri par tas** en $O(n \log(n))$ avec un tas. On peut même utiliser le tableau en entrée comme tableau `tas.t` du tas, ce qui permet d'obtenir un tri en place, c'est-à-dire sans utiliser de tableau supplémentaire (O(1) en mémoire).
- **Algorithme de Dijkstra**, pour extraire le sommet de distance estimée minimum à chaque itération. Dans le pseudocode suivant, on peut utiliser une file de priorité pour `q` :

```
Initialement : q contient tous les sommets
               dist.(s) <- 0
               dist.(v) <- infini, si v <> s

Tant que q est non vide:
    Extraire u de q tel que dist.(u) soit minimum
    Pour tout voisin v de u:
        Si dist.(u) + w(u, v) < dist.(v):
            dist.(v) <- dist.(u) + w(u, v)
```

- **Algorithme de Kruskal**, pour obtenir l'arête de poids minimum à chaque itération.

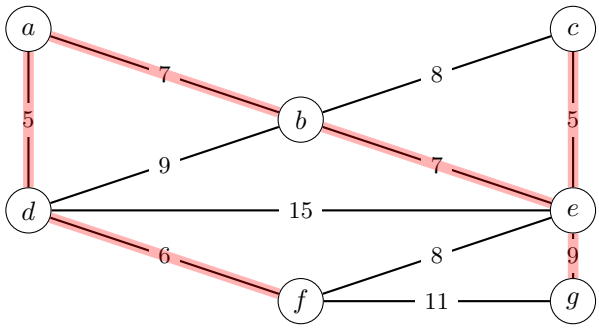
- Soit $G = (V, E)$ un graphe pondéré par $w : E \longrightarrow \mathbb{R}$.
  - Un **arbre couvrant** T de G est un ensemble d'arêtes de G qui forme un arbre et qui contient tous les sommets.
  - Son **poids** $w(T)$ est la somme des poids des arêtes de T.
  - Un arbre couvrant dont le poids est le plus petit possible est appelé un **arbre couvrant de poids minimum**.
- Soit G un graphe connexe pondéré par w.
  Alors G possède un arbre couvrant de poids minimum.

  Preuve : Soit $E = \{w(T) \mid T \text{ est un arbre couvrant de } G\}$.
  - $E \neq \emptyset$: l'ensemble des arêtes parcourues dans un parcours de graphe est un arbre couvrant.
  - E est fini.

  Donc E admet bien un minimum.

  Remarque : il n'y a pas forcément unicité d'un arbre couvrant de poids minimum.
- L'algorithme de **Kruskal** est un algorithme glouton permettant d'obtenir un arbre couvrant de poids minimum en sélectionnant les arêtes par poids croissant qui ne créent pas de cycle.

  Exemple : Un graphe avec un arbre couvrant de poids minimum obtenu par l'algorithme de Kruskal (en choisissant, dans l'ordre : ad, ec, df, ab, be, eg).



- L'algorithme de Kruskal renvoie bien un arbre T qui est couvrant et de poids minimum.

  Preuve : Montrons d'abord que T est un arbre couvrant.
  1. T est acyclique : d'après les choix de l'algorithme.
  2. T est connexe et couvrant : soient u et v deux sommets de G. Soit U l'ensemble des sommets accessibles depuis u dans T.
     Supposons $v \notin U$. Comme G est connexe, il existe une arête de G entre U et $V \setminus U$.
     Cette arête aurait dû être ajoutée à T, puisqu'elle ne créée pas de cycle.
     Contradiction: v est donc accessible depuis u dans T.
     Comme c'est vrai pour tout u, v, T est connexe.

  Montrons maintenant que T est de poids minimum.
  Soient T l'arbre obtenu par Kruskal et T^* un arbre de poids minimum.
  Si $T = T^*$, le théorème est démontré.
  Sinon, soit $e^* = \{u, v\} \in T^* \setminus T$. $T^* - e^*$ n'est pas connexe donc possède deux composantes connexes T^*_u (contenant u) et T^*_v (contenant v).
  Comme T est connexe, il existe un chemin C dans T reliant u et v. Ce chemin possède une arête e entre T^*_u et T^*_v.
  Soit $T^*_2 = T^* - e^* + e$. T^*_2 possède $n-1$ arêtes et une seule composante connexe : c'est un arbre couvrant.
  De plus, $w(e) \leq w(e^*)$ sinon e^* aurait été ajouté avant e dans l'algorithme de Kruskal. On a donc $w(T^*) \geq w(T^*_2)$.
  On répète ce processus avec T^*_2 au lieu de T^*, ce qui nous donne des arbres couvrants T^*_3, T^*_4... jusqu'à obtenir T:

$$w(T^*) \geq w(T^*_2) \geq w(T_3) \geq ... \geq w(T)$$

  $w(T) \leq w(T^*)$ montre que T est de poids minimum.
- Pour l'implémentation, on suppose l'existence d'une fonction `tri_aretes` qui renvoie la liste des arêtes triées par poids croissant (chaque arête étant un triplet (w, u, v) où w est le poids de l'arête $\{u, v\}$).

```
let chemin (t : int list array) u v =
(* détermine s'il existe un chemin de u à v dans t *)
(* on fait un parcours en profondeur depuis u *)
  let n = Array.length t in
  let visited = Array.make n false in
  let rec aux u =
    if not visited.(u) then (
       visited.(u) <- true;
       List.iter aux t.(u)
    ) in
   aux u;
   visited.(v)

let kruskal (g : int list array) =
    let n = Array.length g in
    let t = Array.make n [] in
    List.iter (fun (w, u, v) ->
        if not (chemin t u v) then (
            t.(u) <- v :: t.(u);
            t.(v) <- u :: t.(v);
    ) tri_aretes g;
    done; t
```

Remarque : on peut aussi utiliser une file de priorité au lieu du tri
Complexité : $O(n \log(n))$ (pour le tri ou les n appel la fonction d'extraction d'un tas).

- Un **couplage** de $G = (V, E)$ est un ensemble d'arêtes $M \subseteq E$ tel qu'aucun sommet ne soit adjacent à 2 arêtes de M :

$$\forall e_1, e_2 \in M,\ e_1 \neq e_2 \implies e_1 \cap e_2 = \emptyset$$

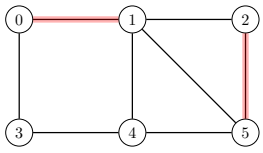
  Un sommet $v \in V$ est **couvert** par M s'il appartient à une arête de M. Sinon, v est **libre** pour M.

- Exercice : Écrire une fonction `est_couplage : (int*int) list -> int > bool` déterminant si une liste d'arêtes (chaque arête étant un couple) forme un couplage.

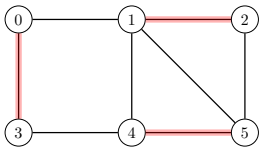
```
let est_couplage m n = (* n = nb de sommets *)
    let couvert = Array.make n false in
    let rec aux l = match l with
        | [] -> true
        | (u, v)::q ->
            if couvert.(u) || couvert.(v) then false
            else begin
                couvert.(u) <- true;
                couvert.(v) <- true;
                aux q
            end
    in aux m
```

- La **taille** de M, notée $|M|$, est son nombre d'arêtes.
  M est un couplage **maximum** s'il n'existe pas d'autre couplage de taille strictement supérieure.
  M est un couplage **maximal** s'Il n'existe pas de couplage M' tel que $M \subsetneq M'$.
  M est un couplage **parfait** si tout sommet de G appartient à une arête de M.
  Un chemin est **élémentaire** s'il ne passe pas deux fois par le même sommet.
  Un chemin élémentaire de G est M-**alternant** si ses arêtes sont alternativement dans M et dans $E \setminus M$.
  Un chemin de G est M-**augmentant** s'il est M-alternant et si ses extrémités sont libres pour M.

- Soit M un couplage de G et P un chemin M-augmentant. Alors $M \Delta P$ est un couplage de G et $|M \Delta P| = |M| + 1$.



Un couplage M



$M \Delta P$, où
$P = 3 - 0 - 1 - 2 - 5 - 4$

- M est un couplage maximum de G $\iff$ Il n'existe pas de chemin M-augmentant dans G.
  Preuve :

$\implies$ Soit M un couplage maximum. Supposons qu'il existe un chemin M-augmentant P. Alors $M \Delta P$ est un couplage de G et $|M \Delta P| > |M|$: absurde.

$\impliedby$ Supposons qu'il existe un couplage M^* vérifiant $|M^*| > |M|$. Considérons $G^* = (V, M \Delta M^*)$.
Les degrés des sommets de G^* sont au plus 2, donc G^* est composé de cycles et de chemins uniquement.
Chacun de ces cycles et chemins alternent entre des arêtes de M et des arêtes de M^*.
Comme $|M^*| > |M|$, un de ces chemins contient plus d'arêtes de M^* que de M : c'est un chemin M-augmentant.

- On en déduit l'algorithme :

Couplage maximum par chemin augmentant

Entrée : Graphe $G = (V, E)$
Sortie : Couplage maximum M de G

$M \leftarrow \emptyset$
Tant que il existe un chemin M-augmentant P dans G **:**
  $M \leftarrow M \Delta P$

Remarques :
- On peut aussi partir d'un couplage initialement non vide, et on obtiendra quand même un couplage maximum à la fin.
- Il est difficile de trouver un chemin M-augmentant dans un graphe quelconque : c'est pour cela qu'on s'intéresse aux graphes bipartis dans la suite.

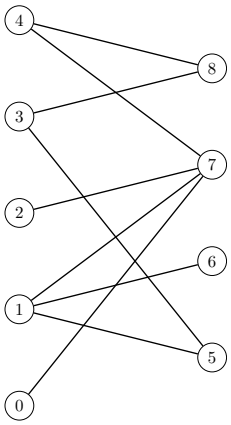
- Un graphe $G = (V, E)$ est **biparti** s'il existe V_1 et V_2 tels que $V = V_1 \sqcup V_2$ et toute arête de E ait une extremité dans V_1 et l'autre dans V_2.
  Remarque : cela revient à donner une couleur à chaque sommet de façon à ce que les extrémités de chaque arête soient de couleurs différentes.

- Exercice : Écrire une fonction `est_biparti : int list array -> bool` pour déterminer si un graphe (représenté par liste d'adjacence) est biparti, en complexité linéaire.
  Solution : On fait un parcours en profondeur depuis un sommet quelconque en alternant les couleurs 0 et 1.

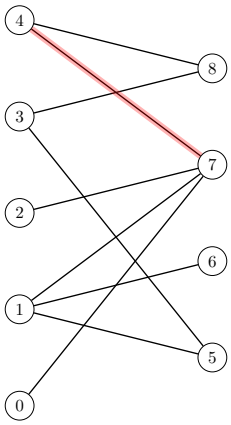
```
let est_biparti g =
    let n = Array.length g in
    let couleurs = Array.make n (-1) in
    let rec aux u c = (* on donne la couleur c à u *)
        if couleurs.(u) = -1 then begin
            couleurs.(u) <- c;
            List.for_all (fun v -> aux v (1 - c)) g.(u)
        end else couleurs.(u) = c
    in aux 0 0
```

- Il est facile de trouver un chemin M-augmentant dans un graphe biparti $G = V_1 \sqcup V_2$:
  - Partir d'un sommet $v \in V_1$ libre.
  - Se déplacer (DFS) en alternant entre des arêtes de M et des arêtes de $G \setminus M$, sans revenir sur un sommet visité.
  - Si on arrive à un sommet libre de V_2, alors on a trouvé un chemin M-augmentant.

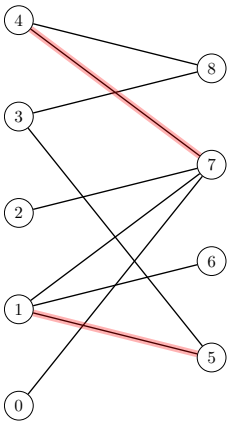
Exemple de recherche d'un couplage maximum par chemin augmentant dans un graphe biparti :



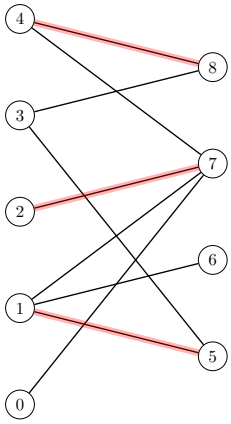
Graphe biparti G et couplage $M = \emptyset$



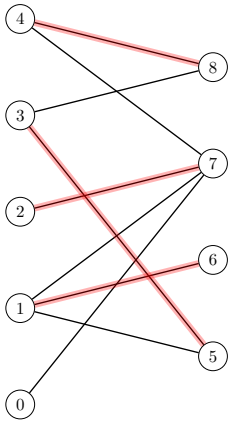
$M \leftarrow M \Delta P$, où $P = 4 - 7$



$M \leftarrow M \Delta P$, où $P = 1 - 5$



$M \leftarrow M \Delta P$, où $P = 2 - 7 - 4 - 8$



$M \leftarrow M \Delta P$, où $P = 3 - 5 - 1 - 6$

- Définitions :

	Signification	Exemple
Alphabet	Ensemble fini de lettres	$\Sigma = \{a, b\}$
Mot	Suite finie de lettre	$m = abaa$
ε	Mot vide (sans lettre)	
Langage	Ensemble de mots	$L = \{\varepsilon, a, baa\}$

ε est un mot, pas une lettre.

- Opérations sur des mots $u = u_1...u_n$ et $v = v_1...v_p$:

	Définition	Exemple avec $u = ab$ et $v = cbc$
Concaténation	$uv = u_1...u_n v_1...v_p$	$uv = abcbc$
Puissance	$u^n = u...u$	$u^3 = ababab$
Taille	$\|u\| = n$	$\|u\| = 2$

Deux mots sont égaux s'ils ont la même taille et les mêmes lettres.

- Opérations sur des langages $L_1 = \{\varepsilon, ab\}$ et $L_2 = \{b, ab\}$:

	Définition	Exemple
Concaténation	$L_1L_2 = \{uv \mid u \in L_1, v \in L_2\}$	$L_1L_2 = \{b, ab, abb, abab\}$
Puissance	$L^n = \{u^n \mid u \in L\}$	$L_1^2 = \{\varepsilon, ab, abab\}$
Etoile	$L^* = \bigcup_{k\in\mathbb{N}} L^k$	$L_1^* = \{\varepsilon, ab, abab...\}$

Comme L_1 et L_2 sont des ensembles, on peut aussi considérer $L_1 \cup L_2$, $L_1 \cap L_2$...

- Les langages rationnels sont tous ceux qu'on peut obtenir avec les règles suivantes :
  - Un langage fini est rationnel
  - L_1 et L_2 rationnels $\implies$ $L_1 \cup L_2$ rationnel
  - L_1 et L_2 rationnels $\implies$ L_1L_2 rationnel
  - L rationnel $\implies$ L^* rationnel

  Exemples : Un alphabet Σ est toujours rationnel car fini. Σ^* est rationnel car est l'étoile du langage rationnel Σ.
- Une expression rationnelle est une suite de symboles contenant : lettres, $\emptyset$, ε, $|$ (union, parfois notée $+$), *.
  À chaque expression rationnelle e on associe un langage $L(e)$.

  Exemple : Le langage de l'expression rationnelle $e = a^*b \mid \varepsilon$ est $L(e) = (\{a\}^*\{b\}) \cup \{\varepsilon\}$.

- L rationnel $\iff$ $\exists$ une expression rationnelle de langage L.
- Définition possible d'expression rationnelle en OCaml :

```
type 'a regexp =
    | Vide | Epsilon | L of 'a
    | Union of 'a regexp * 'a regexp
    | Concat of 'a regexp * 'a regexp
    | Etoile of 'a regexp

(* définition de e ci-dessus *)
let e = Union(Concat(Etoile(a), b), Epsilon)
```

  Exemple : déterminer si ε appartient au langage de e.

```
let rec has_eps e = match e with
    | Vide | L _ -> false
    | Epsilon | Etoile _ -> true
    | Union(e1, e2) -> has_eps e1 || has_eps e2
    | Concat(e1, e2) -> has_eps e1 && has_eps e2
```

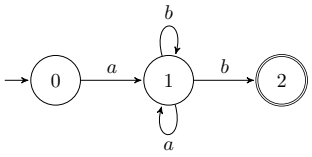
- Quelques techniques de preuve :
  - Sur des mots : récurrence sur la taille du mot.
  - Pour montrer l'égalité de deux langages : double inclusion ou suite d'équivalences.
  - Pour montrer $P(L)$ pour un langage rationnel L : par récurrence, en montrant le cas de base (si L est un langage fini) et les cas d'hérédité ($P(L_1) \wedge P(L_2) \implies P(L_1L_2)$, $P(L_1) \wedge P(L_2) \implies P(L_1 \cup L_2)$, $P(L) \implies P(L^*)$).
  - Pour montrer $P(e)$ pour une expression rationnelle e : par récurrence, en montrant les cas de base ($P(\emptyset)$, $P(\varepsilon)$, $P(a)$, $\forall a \in \Sigma$) et les cas d'hérédité ($P(e_1)$ et $P(e_2) \implies P(e_1e_2)$ et $P(e_1|e_2)$, $P(e) \implies P(e^*)$).

  Exemple : Le miroir d'un mot $u = u_1...u_n$ est $\widetilde{u} = u_n...u_1$ et le miroir d'un langage L est $\widetilde{L} = \{\widetilde{u} \mid u \in L\}$.
  Montrons : L rationnel $\implies$ $\widetilde{L}$ rationnel.
  On pourrait le montrer par récurrence, mais il est peut-être plus simple de définir une fonction $f(e)$ qui à une expression rationnelle e associe une expression rationnelle pour le miroir de $L(e)$:
  - $f(\emptyset) = \emptyset$, $f(\varepsilon) = \varepsilon$ et $\forall a \in \Sigma, f(a) = a$.
  - $f(e_1e_2) = f(e_2)f(e_1)$ (le miroir de uv est $\widetilde{v}\widetilde{u}$).
  - $f(e_1|e_2) = f(e_1)|f(e_2)$.
  - $f(e_1^*) = f(e_1)^*$.

  On a bien défini une fonction f telle que, pour toute expression rationnelle e, $f(e)$ est une expression rationnelle de $\widetilde{L(e)}$. Donc le miroir d'un langage rationnel est rationnel.

- Un **automate** est un 5-uplet $A = (\Sigma, Q, I, F, E)$ où :
  - Σ est un alphabet
  - Q est un ensemble fini d'**états**
  - $I \in Q$ est un ensemble d'**états initiaux**
  - $F \subseteq Q$ est un ensemble d'**états acceptants** (ou **états finaux**)
  - $E \subseteq Q \times \Sigma \times Q$ est un ensemble de **transitions**.
    On peut remplacer l'ensemble E de transitions par une **fonction de transition** $\delta : Q \times \Sigma \longrightarrow \mathcal{P}(Q)$
- – Un chemin dans A est **acceptant** s'il part d'un état initial pour aller dans un état final.
  - Un mot est **accepté** par A s'il est l'étiquette d'un chemin acceptant.
  - Le **langage** $L(A)$ **accepté** (ou **reconnu**) par A est l'ensemble des mots acceptés par A.

  Exemple : le langage $a(a+b)^*b$ est reconnaissable, car reconnu par l'automate ci-dessous.



- Pour déterminer algorithmiquement si un automate A accepte un mot $u = u_1...u_n$, on peut calculer de proche en proche $Q_0 = I$, $Q_1 =$ états accessibles depuis Q_0 avec la lettre u_1, $Q_2 =$ états accessibles depuis Q_0 avec la lettre u_2... et regarder si Q_n contient un état final.

- Soit $A = (\Sigma, Q, I, F, E)$ un automate.
  - A est **complet** si : $\forall q \in Q,\ \forall a \in \Sigma,\ \exists (q, a, q') \in E$
  - Un automate $(\Sigma, Q, \{q_i\}, F, E)$ est **déterministe** si :
    1. Il n'y a qu'un seul état initial q_i.
    2. $(q, a, q_1) \in E \wedge (q, a, q_2) \in E \implies q_1 = q_2$: il y a au plus une transition possible en lisant une lettre depuis un état
  - Un automate déterministe et complet possède une unique transition possible depuis un état en lisant une lettre.
    On a alors $\delta : Q \times \Sigma \longrightarrow Q$ qu'on peut étendre en $\delta^* : Q \times \Sigma^* \longrightarrow Q$ définie par :
    - $\delta^*(q, \varepsilon) = q$
    - Si $u = av$, $\delta^*(q, av) = \delta^*(\delta(q, a), v)$

    On a alors $u \in L(A) \iff \delta^*(q_i, u) \in F$.

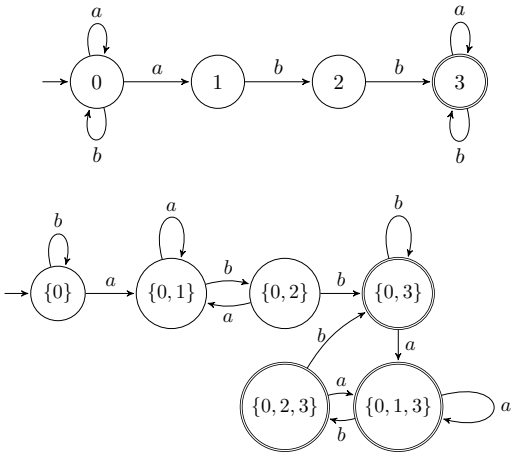
- Deux automates sont **équivalents** s'ils ont le même langage.

- Soit A un automate. Alors A est équivalent à un automate déterministe complet.

  Preuve : Utilise l'automate des parties $A' = (\Sigma, \mathcal{P}(Q), \{I\}, F', \delta')$ où $F' = \{X \subseteq Q \mid X \cap F \neq \emptyset\}$

  Remarque : Si on veut juste un automate complet (pas forcément déterministe), on peut ajouter un état poubelle vers lequel on redirige toutes les transitions manquantes. Dans l'automate des parties, cet état poubelle est $\emptyset$.

  Exemple : Un automate A avec son déterminisé A'.



- Soit L un langage reconnaissable. Alors $\overline{L}$ $(= \Sigma^* \backslash L)$ est reconnaissable.

  Preuve : Soit $A = (\Sigma, Q, q_i, F, \delta)$ un automate déterministe complet reconnaissant L. Alors $A' = (\Sigma, Q, q_i, Q \backslash F, \delta)$ (on inverse états finaux et non-finaux) reconnaît $\overline{L}$.

- Soient L_1 et L_2 des langages reconnaissables. Alors :
  - $L_1 \cap L_2$ est reconnaissable.
  - $L_1 \cup L_2$ est reconnaissable.
  - $L_1 \backslash L_2$ est reconnaissable.

  Preuve : Soient $A_1 = (Q_1, q_1, F_1, \delta_1)$ et $A_2 = (Q_2, q_2, F_2, \delta_2)$ des automates finis déterministes complets reconnaissants L_1 et L_2. Soit $A = (Q_1 \times Q_2, (q_1, q_2), F, \delta)$ (**automate produit**) où :
  - $F = F_1 \times F_2$: A reconnait $L_1 \cap L_2$.
  - $F = \{(q_1, q_2) \mid q_1 \in F_1 \text{ ou } q_2 \in F_2\}$: A reconnait $L_1 \cup L_2$.
  - $F = \{(q_1, q_2) \mid q_1 \in F_1 \text{ et } q_2 \notin F_2\}$: A reconnait $L_1 \setminus L_2$.

  Remarque : Comme l'ensemble des langages reconnaissables est égal à l'ensemble des langages rationnels, l'ensemble des langages rationnels est aussi stable par complémentaire, intersection et différence.
  Il n'y a pas de stabilité par inclusion (L rationnel et $L' \subseteq L$ n'implique pas forcément L' rationnel).

- (**Lemme de l'étoile**) Soit L un langage reconnaissable par un automate à n états.
  Si $u \in L$ et $|u| \geq n$ alors il existe des mots x, y, z tels que :
  - $u = xyz$
  - $|xy| \leq n$
  - $y \neq \varepsilon$
  - $xy^*z \subseteq L$ (c'est-à-dire : $\forall k \in \mathbb{N},\ xy^kz \in L$)

  Application : $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$ n'est pas reconnaissable.
  Preuve : Supposons L_1 reconnaissable par un automate à n états. Soit $u = a^n b^n$. Clairement, $u \in L_1$ et $|u| \geq n$.
  D'après le lemme de l'étoile : il existe x, y, z tels que $u = xyz$, $|xy| \leq n$, $y \neq \varepsilon$ et $xy^*z \subseteq L$.
  Comme $|xy| \leq n$, x et y ne contiennent que des a : $x = a^i$ et $y = a^j$. Comme $y \neq \varepsilon$, $j > 0$.
  En prenant $k = 0$: $xy^0z = xz = a^i b^n \notin L_1$: absurde.

Automate de Glushkov : rationnel $\implies$ reconnaissable

- Une expression rationnelle est **linéaire** si chaque lettre y apparaît au plus une fois : $a(d+c)^*b$ est linéaire mais pas $ac(a+b)$.

- Soit L un langage. On définit :
  - $P(L) = \{a \in \Sigma \mid a\Sigma^* \cap L \neq \emptyset\}$ (premières lettres des mots de L)
  - $S(L) = \{a \in \Sigma \mid \Sigma^* a \cap L \neq \emptyset\}$ (dernières lettres des mots de L)
  - $F(L) = \{u \in \Sigma^2 \mid \Sigma^* u \Sigma^* \cap L \neq \emptyset\}$ (facteurs de longueur 2 des mots de L)
  - L est **local** si, pour tout mot $u = u_1u_2...u_n \neq \varepsilon$:

    $$u \in L \iff u_1 \in P(L) \wedge u_n \in S(L) \wedge \forall k, u_k u_{k+1} \in F(L)$$

    Il suffit donc de regarder la première lettre lettre, la dernière lettre et les facteurs de taille 2 pour savoir si un mot appartient à un langage local.
    Remarques :
    * $\implies$ est toujours vrai donc il suffit de prouver $\impliedby$.
    * Définition équivalente :

      $$L \text{ local} \iff L \setminus \{\varepsilon\} = (P(L) \cap S(L)) \setminus N(L)$$

      où $N(L) = \Sigma^2 \setminus F(L)$.

    Exemples :
    - Si $L_2 = (ab)^*$ alors $P(L_2) = \{a\}$, $S(L_2) = \{b\}$ et $F(L_2) = \{ab, ba\}$. De plus si $u = u_1u_2...u_n \neq \varepsilon$ avec $u_1 \in P(L), u_n \in S(L)$, et $\forall k, u_k u_{k+1} \in F(L)$ alors $u_1 = a$, $u_n = b$ et on montre (par récurrence) que $u = abab...ab \in {}_2$. Donc L_2 est local.
    - Si $L_3 = a^* + (ab)^*$ alors $P(L_3) = \{a\}$, $S(L_3) = \{a, b\}$, $F(L_3) = \{aa, ab, ba\}$. Soit $u = aab$. La première lettre de u est dans $P(L_3)$, la dernière dans $S(L_3)$ et les facteurs de u sont aa et ba qui appartiennent à $F(L_3)$. Mais $u \notin L_3$, ce qui montre que L_3 n'est pas local.

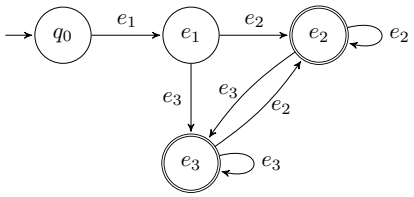
- Un automate déterministe (Σ, Q, q_0, F, E) est **local** si toutes les transitions étiquetées par une même lettre aboutissent au même état : $(q_1, a, q_2) \in E \wedge (q_3, a, q_4) \in E \implies q_2 = q_4$

- Un langage local L est reconnu par un automate local.

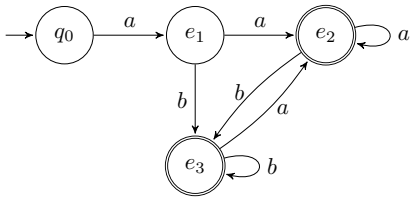
  Preuve : L est reconnu par (Σ, Q, q_0, F, E) où :
  - $Q = \Sigma \cup \{q_0\}$: un état correspond à la dernière lettre lue
  - $F = S(L)$ si $\varepsilon \notin L$, sinon $F = S(L) \cup \{q_0\}$.
  - $E = \{(q_0, a, a) \mid a \in P(L)\} \cup \{(a, b, b) \mid ab \in F(L)\}$

- L'**algorithme de Berry-Sethi** permet de construire un automate à partir d'une expression rationnelle e.

  Exemple avec $e = a(a+b)^*$:
  1. On linéarise e en e', en remplaçant chaque occurrence de lettre dans e par une nouvelle lettre : $e' = e_1(e_2 + e_3)^*$
  2. On peut montrer que $L(e')$ est un langage local.
  3. Un langage local est reconnu par l'automate local $A = (\Sigma, Q, q_0, F, E)$

  4. On fait le remplacement inverse de 1. sur les transitions de A pour obtenir un automate reconnaissant $L(e)$:

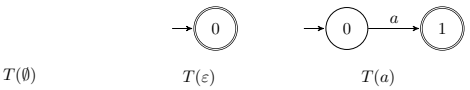
Automate de Thompson : rationnel $\implies$ reconnaissable

- Une ε-transition est une transition étiquetée par ε.

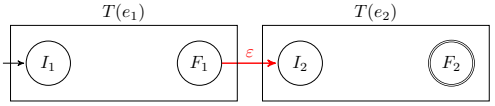
- Un automate avec ε-transitions est équivalent à un automate sans ε-transitions.

  Preuve : Si $A = (\Sigma, Q, I, F, \delta)$ est un automate avec ε-transitions, on définit $A' = (\Sigma, Q, I', F, \delta')$ où :
  - I' est l'ensemble des états atteignables depuis un état de I en utilisant uniquement des ε-transitions.
  - $\delta'(q, a)$ est l'ensemble des états q' tel qu'il existe un chemin de q à q' dans A étiqueté par un a et un nombre quelconque de ε (ce qui peut être trouvé par un parcours de graphe).

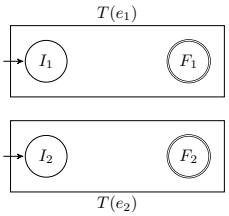
- L'**automate de Thompson** est construit récursivement à partir d'une expression rationnelle e :
  - Cas de base :

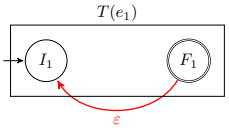
    $T(\emptyset)$ $T(\varepsilon)$ $T(a)$
  - $T(e_1e_2)$: ajout d'une ε-transition depuis chaque état final de $T(e_1)$ vers chaque état initial de $T(e_2)$.

  - $T(e_1|e_2)$: union des états initiaux et des états finaux.



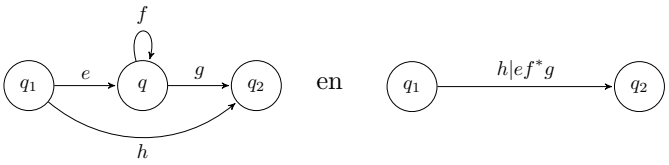
– $T(e_1^*)$: ajout d'une ε-transition depuis chaque état final vers chaque état initial.



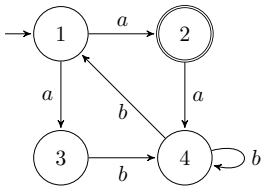
Élimination des états : reconnaissable $\Longrightarrow$ rationnel

- Tout automate est équivalent à un automate avec un unique état initial sans transition entrante et un unique état final sans transition sortante.
  Preuve : On ajoute un état initial q_i et un état final q_f et des transitions ε depuis q_i vers les états initiaux et depuis les états finaux vers q_f.

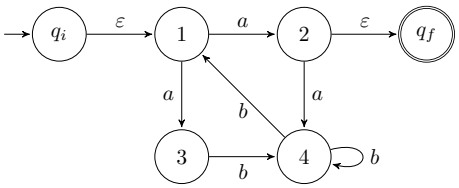
- **Méthode d'élimination des états** : On considère un automate A comme dans le point précédent. Tant que A possède au moins 3 états, on choisit un état $q \notin \{q_i, q_f\}$ et on supprime q en modifiant les transitions :



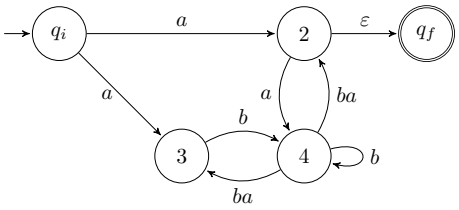
Exemple :



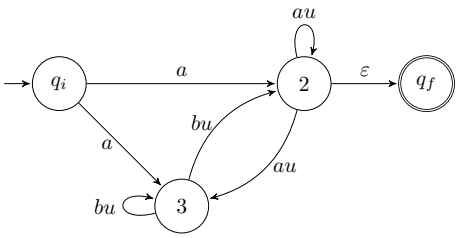
1. On commence par se ramener à un automate avec un état initial sans transition entrante et un état final sans transition sortante :



2. Suppression de l'état 1 :

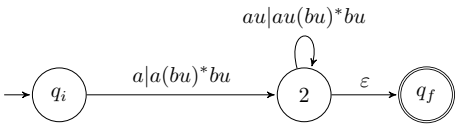


3. Suppression de l'état 4 :

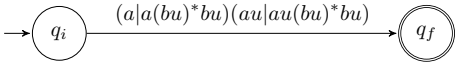


Avec $u = b^*ba$.

4. Suppression de l'état 3 :



5. Suppression de l'état 2 :



On obtient l'expression rationnelle $a|a(bu)^*bu(au|au(bu)^*bu)$, où $u = b^*ba$.

Rationnel $\Longleftrightarrow$ reconnaissable

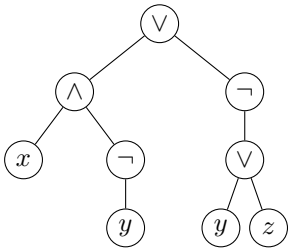
- **Théorème de Kleene** : un langage est rationnel si et seulement si il est reconnaissable par un automate.

- Les théorèmes sur les automates s'appliquent aussi aux langages rationnels, et inversement. Notamment, les langages rationnels sont stables par union, concaténation, étoile, intersection, complémentaire, différence.

- Soit V un ensemble (de **variables**). L'ensemble des **formules logiques** sur V est défini inductivement :
  - T et F sont des formules (Vrai et Faux)
  - Toute variable $x \in V$ est une formule
  - Si φ est une formule alors $\neg\varphi$ est une formule
  - Si φ, ψ sont des formules alors $\varphi \wedge \psi$ (conjonction) et $\varphi \vee \psi$ (disjonction) sont des formules

```
type 'a formula =
    | T | F (* true, false *)
    | Var of 'a (* variable *)
    | Not of 'a formula
    | And of 'a formula * 'a formula
    | Or of 'a formula * 'a formula
```

- On peut représenter une formule logique par un arbre.
  Exemple : $(x \wedge \neg y) \vee \neg(y \vee z)$ est représenté par



  - L'**arité** d'un connecteur logique est son nombre d'arguments (= nombre de fils dans l'arbre).
    $\neg$ est d'arité 1 (unaire) et $\wedge, \vee$ sont d'arités 2 (binaire).
  - La **taille** d'une formule est le nombre de symboles qu'elle contient (= nombre de noeuds de l'arbre).
  - La **hauteur** d'une formule est la hauteur de l'arbre associé.

- (Exemple de démonstration par induction sur les formules) Soit φ une formule ayant $n(\varphi)$ symboles de négation et $b(\varphi)$ connecteurs binaire. Alors la taille $t(\varphi)$ de φ est : $t(\varphi) = 1 + n(\varphi) + 2b(\varphi)$.

  Preuve : Montrons $P(\varphi) : t(\varphi) = 1 + n(\varphi) + 2b(\varphi)$ par induction.
  Cas de base : $t(T) = 1 = 1 + 0 + 0$ donc $P(T)$ est vraie.
  De même pour $P(F)$ et $P(x)$ où x est une variable.
  Hérédité : Soit φ une formule.
  - Si $\varphi = \neg\psi$ alors $t(\psi) = 1 + n(\psi) + 2b(\psi)$ par induction et $t(\varphi) = t(\neg\psi) = 1 + t(\psi) = 1 + \underbrace{1 + n(\psi)}_{n(\varphi)} + 2\underbrace{b(\psi)}_{b(\varphi)} =$ $1 + n(\varphi) + 2b(\varphi)$ donc $P(\varphi)$ est vraie.
  - Si $\varphi = \psi_1 \wedge \psi_2$ alors, par induction, $t(\psi_1) = 1 + n(\psi_1) + 2b(\psi_1)$ et $t(\psi_2) = 1 + n(\psi_2) + 2b(\psi_2)$. Donc $t(\varphi) = 1 + t(\psi_1) + t(\psi_2) = 1 + \underbrace{n(\psi_1) + n(\psi_2)}_{n(\varphi)} + 2\underbrace{(1 + b(\psi_1) + b(\psi_2))}_{b(\varphi)}$.
    Donc $P(\varphi)$ est vraie.
  - De même si $\varphi = \psi_1 \vee \psi_2$.

  Par induction structurelle, $P(\varphi)$ est donc vraie pour toute formule φ.

- $\varphi \longrightarrow \psi$ est défini par $\neg\varphi \vee \psi$.
  $\varphi \longleftrightarrow \psi$ est défini par $\varphi \longrightarrow \psi \wedge \psi \longrightarrow \varphi$.

- Une **valuation** sur un ensemble V de variables est une fonction $v : V \longrightarrow \{0, 1\}$. 0 est aussi noté Faux ou $\bot$. 1 est aussi noté Vrai ou $\top$. L'**évaluation** $[\![\varphi]\!]_v$ d'une formule φ sur v est définie inductivement :
  - $[\![T]\!]_v = 1$, $[\![F]\!]_v = 0$
  - $[\![x]\!]_v = v(x)$ si $x \in V$
  - $[\![\neg\varphi]\!]_v = 1 - [\![\varphi]\!]_v$
  - $[\![\varphi \wedge \psi]\!]_v = \min([\![\varphi]\!]_v, [\![\psi]\!]_v)$
  - $[\![\varphi \vee \psi]\!]_v = \max([\![\varphi]\!]_v, [\![\psi]\!]_v)$

  Si $[\![\varphi]\!]_v = 1$, on dit que v est un **modèle** pour φ.

```
let rec eval d = function
    | T -> true
    | F -> false
    | Var(x) -> d x
    | Not(p) -> not (eval p)
    | And(p, q) -> (eval p) && (eval q)
    | Or(p, q) -> (eval p) || (eval q)
```

- Une formule toujours évaluée à 1 est une **tautologie**. Une formule toujours évaluée à 0 est une **antilogie**. Une formule qui possède au moins une évaluation à 1 est **satisfiable**.

- Deux formules φ et ψ sur V sont **équivalentes** (et on note $\varphi \equiv \psi$) si, pour toute valuation $v : V \to \{0, 1\}$: $[\![\varphi]\!]_v = [\![\psi]\!]_v$.
  - $\neg\neg\varphi \equiv \varphi$
  - $\varphi \vee \neg\varphi \equiv T$ (toujours vrai)
  - $\neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$ (de Morgan)
  - $\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi$ (de Morgan)
  - $\varphi_1 \vee (\varphi_2 \wedge \varphi_3) \equiv (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$
  - $\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \equiv (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$

- La table de vérité permet de voir rapidement quelles sont les évaluations possibles d'une formule. Une formule à n variables possède 2^n évaluations possibles, et donc 2^n lignes dans sa table de vérité.

x	y	$(x \wedge y) \vee (\neg x \wedge \neg y)$
0	0	1
0	1	0
1	0	0
1	1	1

Table de vérité de $(x \wedge y) \vee (\neg x \wedge \neg y)$

- La logique propositionnelle définit la notion de formule vraie (si elle est vraie pour toute valuation).
  La déduction naturelle permet de formaliser la notion de preuve mathématique.
- Un **séquent** est noté $\Gamma \vdash A$ où Γ est un ensemble de formules logiques et A une formule logique.
  $\Gamma \vdash A$ signifie Intuitivement que sous les hypothèses Γ, on peut déduire A.
- Règles de déduction naturelle classique, où A, B, C sont des formules quelconques :

	Introduction	**Élimination**
Conjonction	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge_i$	$\dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge_e^g \qquad \dfrac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge_e^d$
Disjonction	$\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee_i^g \qquad \dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee_i^d$	$\dfrac{\Gamma, A \vdash C \quad \Gamma, B \vdash C \quad \Gamma \vdash A \vee B}{\Gamma \vdash C}\ \vee_e$
Implication	$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ \to_i$	$\dfrac{\Gamma \vdash A \to B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \to_e$
Négation	$\dfrac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg_i$	$\dfrac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \neg_e$
Vrai $\top$	$\dfrac{}{\Gamma \vdash \top}\ \top_i$	
Faux $\bot$		$\dfrac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot_e$

Axiome	**Affaiblissement**	**Réduction à l'absurde**
$\dfrac{}{\Gamma, A \vdash A}\ \text{ax}$	$\dfrac{\Gamma \vdash A}{\Gamma, B \vdash A}\ \text{aff}$	$\dfrac{\Gamma, \neg A \vdash \bot}{\Gamma \vdash A}\ \text{raa}$

Il n'est pas nécessaire d'apprendre ces règles par coeur (elles seront rappelées), mais il faut les comprendre et savoir les utiliser.

- Une **preuve** d'un séquent $\Gamma \vdash A$ est un arbre dont les nœuds sont des séquents, les arcs des règles et la racine est $\Gamma \vdash A$.
  Exemples :

1. Preuve de $(A \wedge B) \to C \vdash A \to (B \to C)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \wedge B \to C \vdash A \wedge B \to C}\ \text{ax} \quad \dfrac{\dfrac{}{A \vdash A}\ \text{ax} \quad \dfrac{}{B \vdash B}\ \text{ax}}{A, B \vdash A \wedge B}\ \wedge_i}{(A \wedge B) \to C, A, B \vdash C}\ \to_e}{(A \wedge B) \to C, A \vdash B \to C}\ \to_i}{(A \wedge B) \to C \vdash A \to (B \to C)}\ \to_i$$

2. Preuve de $A \vdash \neg\neg A$:

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\ \text{ax} \quad \dfrac{}{\neg A \vdash \neg A}\ \text{ax}}{A, \neg A \vdash \bot}\ \neg_e}{A \vdash \neg\neg A}\ \neg_i$$

3. On peut décomposer une preuve longue en plusieurs parties, pour plus de lisibilité. Par exemple pour prouver $\vdash A \vee (B \wedge C) \longrightarrow (A \vee B) \wedge (A \vee C)$:

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\text{ax}}{A \vdash A \vee B}\,\vee_i^g \qquad \dfrac{\dfrac{\dfrac{}{B \wedge C \vdash B \wedge C}\,\text{ax}}{B \wedge C \vdash B}\,\wedge_e^g}{B \wedge C \vdash A \vee B}\,\vee_i^d \qquad \dfrac{}{A \vee (B \wedge C) \vdash A \vee (B \wedge C)}\,\text{ax}}{A \vee (B \wedge C) \vdash A \vee B\ (*)}\,\vee_e$$

On montre de même $A \vee (B \wedge C) \vdash A \vee C$ $(**)$ et finalement :

$$\dfrac{\dfrac{\dfrac{}{A \vee (B \wedge C) \vdash A \vee B}\,* \qquad \dfrac{}{A \vee (B \wedge C) \vdash A \vee C}\,**}{A \vee (B \wedge C) \vdash (A \vee B) \wedge (A \vee C)}\,\wedge_i}{\vdash A \vee (B \wedge C) \longrightarrow (A \vee B) \wedge (A \vee C)}\,\longrightarrow_i$$

- (Correction de la déduction naturelle) Si $\Gamma \vdash A$ est prouvable alors $\Gamma \models A$.

Preuve : Soit $P(h)$: « si T est un arbre de preuve de hauteur h pour $\Gamma \vdash A$ alors $\Gamma \models A$ ».
$P(0)$ est vraie : Si T est un arbre de hauteur 0 pour $\Gamma \models A$ alors il est constitué uniquement d'une application de ax, ce qui signifie que $A \in \Gamma$ et implique $\Gamma \models A$.
Soit T un arbre de preuve pour pour $\Gamma \vdash A$ de hauteur $h+1$. Considérons la règle appliquée à la racine de T.

– $(\wedge_i)$ Supposons T de la forme : $\dfrac{\dfrac{T_1}{\Gamma \vdash A} \qquad \dfrac{T_2}{\Gamma \vdash B}}{\Gamma \vdash A \wedge B}\,\wedge_i$
Par hypothèse de récurrence sur T_1 et T_2, on obtient $\Gamma \models A$ et $\Gamma \models B$.
Une valuation v satisfaisant toutes les formules de Γ satisfait donc à la fois A et B, et donc $A \wedge B$. On a bien $\Gamma \models A \wedge B$.

– $(\wedge_e)$ Supposons T de la forme : $\dfrac{\dfrac{T_1}{\Gamma \vdash A \wedge B}}{\Gamma \vdash A}\,(\wedge_e^g)$ Par récurrence sur T_1, $\Gamma \models A \wedge B$ et donc $\Gamma \models A \wedge B$.

– Les autres cas sont similaires...

1 Programme de première année

1.1 Programmation récursive

La capacité d'un programme à faire appel à lui-même est un concept primordial en informatique. Historiquement, l'auto-référence est au cœur du paradigme de programmation fonctionnelle. Elle imprègne aujourd'hui, de manière plus ou moins marquée, la plupart des langages de programmation contemporains. Elle permet souvent d'écrire des algorithmes plus élégants et moins laborieux que leurs équivalents en programmation impérative.

Notions	**Commentaires**
Récursivité. Récursivité croisée. Organisation des activations sous forme d'arbre en cas d'appels multiples.	On approfondit la présentation purement expérimentale de l'appel récursif d'une fonction à elle-même, vue au premier semestre de tronc commun, en faisant le lien avec le principe de récurrence et les relations d'ordre. On insiste sur le problème de la terminaison et la notion d'ordre bien fondé. Toute théorie générale de la dérécursification est hors programme.
Stratégie diviser pour régner.	On complète, si nécessaire, la présentation de tris élémentaires vue au premier semestre de tronc commun en présentant le tri par partition fusion. On ne se limite pas à des exemples dans lesquels la décomposition et la recomposition des résultats sont évidents.
Analyse de la complexité des algorithmes récursifs.	On étudie sur différents exemples la complexité dans le pire cas. Aucun théorème général de complexité n'est au programme. Sur des exemples, on sensibilise les étudiants à l'existence d'autres analyses de complexité, comme la complexité en moyenne ou la complexité amortie. La notion de classe de complexité d'un problème de décision est hors programme.

1.2 Types récursifs immuables et arbres

Les propriétés sur les structures récursives sont démontrées par des preuves par induction structurelle.

Notions	**Commentaires**
Type récursif de liste.	

Type récursif d'arbre binaire ou d'arbre binaire strict non vide.
 	
Vocabulaire : nœud, nœud interne, racine, feuille, fils, père, hauteur d'un arbre, profondeur d'un nœud, étiquette, sous-arbre.
	`type 'a arbre_binaire = Vide | Noeud of 'a * 'a arbre_binaire * 'a arbre_binaire`
	`type ('a, 'b) abs = Feuille of 'a | Noeud_interne of 'b * ('a, 'b) abs * ('a, 'b) abs`
	La hauteur de l'arbre vide est -1. Relation entre le nombre de nœuds internes et de feuilles d'un arbre binaire strict.

Définition inductive des arbres généraux non vides.	On illustre la notion d'arbre par des exemples, comme : expression arithmétique, arbre préfixe (*trie*), arbre de décision.
Parcours d'arbres. Ordre préfixe, infixe et postfixe.	On peut évoquer le lien avec l'empilement de blocs d'activation lors de l'appel à une fonction récursive.

1.3 Structures de données

Le programme de l'option MP se distingue du tronc commun en ce qu'il appelle la maîtrise du concept de structure de données : il dépasse l'idée que le langage de programmation fournisse une collection appropriée à ses besoins et insiste sur le fait que le développement d'un algorithme aille de pair avec la conception d'une structure de données taillée à la mesure du problème que l'on cherche à résoudre et des opérations sur les données que l'on est amené à répéter. Bien que la programmation orientée objet ne figure pas dans ce programme, on peut enseigner la notion de structure de données avec cette perspective à l'esprit.

Notions	Commentaires
Définition d'une structure de données abstraite comme un type muni d'opérations.	On parle de constructeur pour l'initialisation d'une structure, d'accesseur pour récupérer une valeur et de transformateur pour modifier l'état de la structure. On montre l'intérêt d'une structure de données abstraite en terme de modularité. On distingue la notion de structure de données abstraite de son implémentation. Plusieurs implémentations concrètes sont interchangeables.
Distinction entre structure de données mutable et immuable. Référence.	
Tableau, liste, pile, file, dictionnaire.	On complète la présentation purement pratique faite de ces structures dans le cadre du programme de tronc commun en étudiant dans le détail leurs implémentations possibles et en s'attachant à dégager la complexité des opérations associées. On clarifie l'ambiguïté du terme *liste* utilisé en tant que concept en informatique, tableau dynamique dans le langage Python et chaîne de maillons dans le langage OCaml. On présente des problèmes qui peuvent être résolus à l'aide de ces structures. On peut présenter différentes implémentations de la même structure.
Implémentation de la structure immuable de dictionnaire avec un arbre binaire de recherche.	On ne cherche pas à équilibrer les arbres. En lien avec le programme de tronc commun de deuxième année, on peut présenter également une implémentation mutable des dictionnaires avec une table de hachage.
Utilisation d'une structure de données.	Grâce aux bibliothèques, on peut utiliser des structures de données sans avoir à programmer soi-même leur implémentation.

1.4 Syntaxe puis sémantique de la logique propositionnelle

Le but de cette partie est de familiariser progressivement les étudiants avec la différence entre syntaxe et sémantique d'une part et de donner une base de vocabulaire permettant de modéliser une grande variété de situations (par exemple, satisfaction de contraintes, planification, diagnostique, vérification de modèles, etc.).
L'étude des quantificateurs est hors programme.

Notions	Commentaires
Variables propositionnelles, connecteurs logiques, arité.	Notations : $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$.
Formules propositionnelles, définition inductive, représentation comme un arbre. Sous-formule. Taille et hauteur d'une formule.	Les formules sont des données informatiques. On fait le lien entre les écritures d'une formule comme mot et les parcours d'arbres.
Valuations, valeurs de vérité d'une formule propositionnelle.	Notations V pour la valeur vraie, F pour la valeur fausse.
Satisfiabilité, modèle, ensemble de modèles, tautologie, antilogie.	Une formule est satisfiable si elle admet un modèle, tautologique si toute valuation en est un modèle. On peut être amené à ajouter à la syntaxe une formule tautologique et une formule antilogique ; elles sont en ce cas notées $\top$ et $\bot$.
Équivalence sur les formules.	On présente les lois de De Morgan, le tiers exclu et la décomposition de l'implication.
Conséquence logique entre deux formules.	On étend la notion à celle de conséquence ϕ d'un ensemble de formules Γ : on note $\Gamma \models \phi$. La compacité est hors programme.
Forme normale conjonctive, forme normale disjonctive. Problème SAT.	Lien entre forme normale disjonctive complète et table de vérité.

2 Programme de deuxième année

2.1 Applications des arbres

On aborde les arbres comme support de pensée qui permet de donner du sens et de raisonner sur le flot de contrôle qui s'observe dans la mise en œuvre de stratégies algorithmiques ou de structures de données élaborées.

Notions	**Commentaires**
Retour sur trace (*backtracking*).	On présente la notion à travers quelques exemples sans théorie générale (par exemple, problème des huit reines ou résolution d'un sudoku). La notion de retour sur trace est réinvestie par l'étudiant à l'occasion de l'étude de l'algorithme min-max présenté en tronc commun.
File de priorité.	Implémentation de la structure mutable de file de priorité bornée à l'aide d'un tas stocké dans un tableau. On illustre l'intérêt d'une file de priorité pour améliorer l'implémentation sommaire de l'algorithme de Dijkstra vue dans le cadre du programme de tronc commun. On présente le tri par tas d'un tableau. Pour l'algorithme de Kruskal, l'utilisation d'une file de priorité est une alternative intéressante au tri des arêtes lorsqu'elles sont stockées dans un tableau.

2.2 Graphes avancés

Le vocabulaire et les définitions relatives aux graphes, orientés ou non, tels que sommets, arcs, arêtes, degrés, ont été présentés dans le cadre du programme d'informatique de tronc commun.

Notions	**Commentaires**
Notion de parcours (sans contrainte). Parcours en largeur, en profondeur. Notion d'arborescence d'un parcours.	On approfondit la notion de parcours de graphe dont l'étude a été entamée dans le cadre du programme de tronc commun et on en détaille quelques applications : par exemple, recherche et construction d'un cycle, calcul des composantes connexes, bicolorabilité, états accessibles d'un automate. On étudie l'intérêt d'un type de parcours dans le cadre d'applications. Par exemple : plus courts chemins dans un graphe à distance unitaire ou tri topologique dans un graphe acyclique orienté. On implémente ces parcours à l'aide d'une représentation du graphe en listes d'adjacence et on discute leur complexité.
Arbre couvrant dans un graphe pondéré.	Algorithme de Kruskal. Les détails d'implémentation sont laissés à l'appréciation du professeur. On fait le lien avec la notion d'algorithme glouton étudiée dans le programme de tronc commun.
Graphe biparti. Couplage.	Recherche d'un couplage de cardinal maximum dans un graphe biparti par des chemins augmentants. On se limite à une approche élémentaire. L'algorithme de Hopcroft-Karp n'est pas au programme.

2.3 Déduction naturelle pour la logique propositionnelle

Il s'agit de présenter les preuves comme permettant de pallier deux problèmes de la présentation du calcul propositionnel faite en première année : nature exponentielle de la vérification d'une tautologie, faible lien avec les preuves mathématiques.
Il ne s'agit, en revanche, que d'introduire la notion d'arbre de preuve. La déduction naturelle est présentée comme un jeu de règles d'inférence simple permettant de faire un calcul plus efficace que l'étude de la table de vérité. Toute technicité dans les preuves dans ce système est à proscrire.
On s'abstient d'implémenter ces règles. L'ambition est d'être capable d'écrire de petites preuves dans ce système.

Notions	Commentaires
Déduction naturelle. Règle d'inférence, dérivation. Définition inductive d'un arbre de preuve.	Notation $\vdash$. Séquent $H_1, \dots, H_n \vdash C$. On présente des exemples tels que le *modus ponens* $(p, p \to q \vdash q)$ ou le syllogisme *barbara* $(p \to q, q \to r \vdash p \to r)$. On présente des exemples utilisant les règles précédentes.
Règles d'introduction et d'élimination de la déduction naturelle pour les formules propositionnelles. Correction de la déduction naturelle pour les formules propositionnelles.	On présente les règles pour $\wedge, \vee, \neg$ et $\to$. On écrit de petits exemples d'arbre de preuves (par exemple $\vdash (p \to q) \to \neg(p \wedge \neg q)$, etc.).

2.4 Langages et automates

On introduit les expressions régulières comme formalisme dénotationnel pour spécifier un motif dans le cadre d'une recherche textuelle et les automates comme formalisme opérationnel efficace pour la recherche de motifs. On vérifie que le formalisme des automates coïncide exactement avec l'expressivité des expressions régulières.

Notions	Commentaires
Alphabet, mot, préfixe, suffixe, facteur, sous-mot. Langage comme ensemble de mots sur un alphabet.	Le mot vide est noté ε.
Opérations régulières sur les langages (union, concaténation, étoile de Kleene). Définition inductive des langages réguliers. Expression régulière. Dénotation d'une expression régulière.	On introduit les expressions régulières comme un formalisme dénotationnel pour les motifs. On note l'expression dénotant le langage vide $\emptyset$, celle dénotant le langage réduit au mot vide ε, l'union par $\vert$, la concaténation par juxtaposition et l'étoile de Kleene par une étoile.
Automate fini déterministe. État accessible, co-accessible. Automate émondé ; automate complet. Langage reconnu par un automate.	
Automate fini non déterministe. Transition spontanée (ou ε-transition). Déterminisation d'un automate non déterministe.	On aborde l'élimination des ε-transitions et plus généralement les constructions d'automates à la Thompson sur des exemples, sans chercher à formaliser complètement les algorithmes sous-jacents.
Construction de l'automate de Glushkov associé à une expression régulière par l'algorithme de Berry-Sethi.	Les notions de langage local et d'expression régulière linéaire sont introduites dans cette seule perspective.
Passage d'un automate à une expression régulière. Élimination des états. Théorème de Kleene.	On se limite à la description du procédé d'élimination et à sa mise en œuvre sur des exemples d'automates de petite taille ; cela constitue la preuve du sens réciproque du théorème de Kleene.
Stabilité de la classe des langages reconnaissables par union finie, intersection finie, complémentaire.	
Lemme de l'étoile.	Soit L le langage reconnu par un automate à n états : pour tout $u \in L$ tel que $\vert u \vert \geq n$, il existe x, y, z tels que $u = xyz$, $\vert xy \vert \leq n$, $y \neq \varepsilon$ et $xy^*z \subseteq L$.

A Langage OCaml

La présente annexe liste limitativement les éléments du langage OCaml (version 4 ou supérieure) dont la connaissance, selon les modalités de chaque sous-section, est exigible des étudiants. Aucun concept sous-jacent n'est exigible au titre de la présente annexe.

A.1 Traits et éléments techniques à connaitre

Les éléments et notations suivants du langage OCaml doivent pouvoir être compris et utilisés par les étudiants dès la fin de la première année sans faire l'objet d'un rappel, y compris lorsqu'ils n'ont pas accès à un ordinateur.

Traits généraux

— Typage statique, inférence des types par le compilateur. Idée naïve du polymorphisme.
— Passage par valeur.
— Portée lexicale : lorsqu'une définition utilise une variable globale, c'est la valeur de cette variable au moment de la définition qui est prise en compte.
— Curryfication des fonctions. Fonction d'ordre supérieur.
— Gestion automatique de la mémoire.
— Les retours à la ligne et l'indentation ne sont pas signifiants mais sont nécessaires pour la lisibilité du code.

Définitions et types de base

— `let`, `let rec` (pour des fonctions), `let rec` … `and` …. `fun` x y `->` e.
— `let` v `=` e `in` e', `let rec` f x `=` e `in` e'.
— Expression conditionnelle `if` e `then` e_V `else` e_F.
— Types de base : `int` et les opérateurs `+`, `-`, `*`, `/`, l'opérateur `mod` quand toutes les grandeurs sont positives ; `float` et les opérateurs `+.`, `-.`, `*.`, `/.` ; `bool`, les constantes `true` et `false` et les opérateurs `not`, `&&`, `||` (y compris évaluation paresseuse). Entiers et flottants sont sujets aux dépassements de capacité.
— Comparaisons sur les types de base : `=`, `<>`, `<`, `>`, `<=`, `>=`.

Types structurés

— n-uplets ; non-nécessité d'un `match` pour récupérer les valeurs d'un n-uplet.
— Listes : type `'a list`, constructeurs `[]` et `::`, notation [x ; y ; z] ; opérateur `@` (y compris sa complexité) ; `List.length`. Motifs de filtrage associés.
— Type `'a option`.
— Déclaration de type, y compris polymorphe.
— Types énumérés (ou sommes, ou unions), récursifs ou non ; les constructeurs commencent par une majuscule, contrairement aux identifiants. Motifs de filtrage associés.
— Filtrage : `match` e `with` p_0 `->` v_0 `|` p_1 `->` v_1 … ; les motifs sont exhaustifs, ils ne doivent pas comporter de variable utilisée antérieurement ni deux fois la même variable ; motifs plus ou moins généraux, notation `_`, importance de l'ordre des motifs quand ils ont des instances communes.

Programmation impérative

— Absence d'instruction ; la programmation impérative est mise en œuvre par des expressions impures ; `unit`, `()`.
— Références : type `'a ref`, notations `ref`, `!`, `:=`. Les références doivent être utilisées à bon escient.
— Séquence `;`. La séquence intervient entre deux expressions.
— Boucle `while` c `do` b `done` ; boucle `for` v `=` d `to` f `do` b `done` (on rappelle, quand cela est utile au problème étudié, que les deux bornes sont atteintes).

Divers

— Usage de `begin` … `end`.

— Exceptions : `failwith`.
— Utilisation d'un module : notation $M.f$. Les noms des modules commencent par une majuscule.
— Syntaxe des commentaires, à l'exclusion de la nécessité d'équilibrer les délimiteurs dans un commentaire.

A.2 Éléments techniques devant être reconnus et utilisables après rappel

Les éléments suivants du langage OCaml doivent pouvoir être utilisés par les étudiants pour écrire des programmes dès lors qu'ils ont fait l'objet d'un rappel et que la documentation correspondante est fournie.

Définition et types de base

— Types de base : opérateur `mod` avec opérandes de signes quelconques, opérateur `**`.
— Types `char` et `string` ; `'`x`'` quand x est un caractère imprimable, `"`x`"` quand x est constituée de caractères imprimables, `String.length`, s`.[`i`]`, opérateur `^`. Existence d'une relation d'ordre total sur `char`. Immuabilité des chaines.
— Fonctions de conversion entre types de base.
— `print_int`, `print_string`, `print_float`.

Types structurés et structures de données

— Listes : les fonctions `mem`, `exists`, `for_all`, `filter`, `map` et `iter` du module `List`.
— Tableaux : type `'a array`, notations `[|...|]`, t`.(`i`)`, t`.(`i`) <-` v ; les fonctions suivantes du module `Array` : `length`, `make`, `make_matrix`, `init`, `copy`, `mem`, `exists`, `for_all`, `map` et `iter`.
— Types enregistrements immuables et mutables, notations associées.
— Types mutuellement récursifs.
— Piles et files mutables : fonctions `create`, `is_empty`, `push` et `pop` des modules `Queue` et `Stack`.
— Dictionnaires mutables réalisés par tables de hachage sans liaison multiple ni randomisation par le module `Hashtbl` : fonctions `create` (on élude toute considération sur la taille initiale), `add`, `remove`, `mem`, `find`, `find_opt` et `iter`.