DS2 (sujet type Mines Centrale)

1 Problème sur les arbres couvrants (C)

Dans ce problème, un graphe G est un couple (S, A) où :

- S est un ensemble fini dont les éléments sont les sommets de G ;
- $A = (a_0, a_1, \dots, a_{m-1})$ est la suite des arêtes de G, une arête étant une partie $a = \{s, t\}$ de S de cardinal 2. Les sommets s et t sont appelés les extrémités de l'arête a et on dira que a relie s et t. Si s et t sont reliés par une arête, on dit qu'ils sont voisins ou adjacents.

Ainsi, les graphes sont non orientés et il n'y a pas d'arête reliant un sommet à lui-même.
Par convention, nous noterons n (respectivement m) le nombre de sommets (respectivement d'arêtes) du graphe et nous supposerons que $S = \{0, 1, \dots, n-1\} = S_n$.

Un graphe sera représenté par le type C suivant :

```
struct graphe = {int size;
                 int* degrees;
                 int** adj;};

typedef struct graphe graphe;
```

Si G est un graphe représenté par `g`, alors le nombre de sommets `n` du graphe est donné par le champs size, le tableau `g.degrees` de taille n contient dans sa case d'indice `i` le degré du sommet `i` et le tableau `g.adj[i]` est de taille `g.degrees[i]` et contient les voisins du sommet `i` dans G.

Soit $G = (S, A)$ un graphe.

- Un chemin dans G est une suite $c = (s_0, s_1, \dots, s_{j-1}, s_j, \dots, s_{k-1}, s_k)$ où pour tout j compris entre 1 et k, s_{j-1} et s_j sont des sommets voisins. On dira que c est un chemin de s_0 à s_k de longueur k. Par convention, pour s sommet de G, il existe un chemin de longueur nulle de s à s.
- La composante connexe d'un sommet s de G, notée C_s , est l'ensemble des sommets t de G tels qu'il existe un chemin de s à t.
- On dit que G est connexe si pour tous sommets s et t de G, il existe un chemin de s à t.
- Un cycle dans G est un chemin de longueur $k \geq 2$ d'un sommet à lui-même et dont les arêtes sont deux à deux distinctes. On dit que G est acyclique s'il ne contient aucun cycle.
- Un arbre est un graphe connexe acyclique.

1.1 Caractérisation des arbres

Soit $G = (S_n, A)$ un graphe à n sommets et m arêtes, représenté par `g`. Les arêtes de G sont donc numérotées $A = (a_0, a_1, \dots, a_{m-1})$.

Q1 Montrer que les composantes connexes de G forment une partition de S_n , c'est-à-dire que :

1. $\forall s \in S_n, C_s \neq \emptyset$;
2. $S_n = \bigcup_{s \in S_n} C_s$;
3. Pour tous sommets s et t, soit $C_s = C_t$, soit $C_s \cap C_t = \emptyset$.

Q2 Montrer que si s et t sont deux sommets de G tels que $t \in C_s$, il existe un plus court chemin de s à t et que les sommets d'un plus court chemin sont deux à deux distincts.

Pour $k \in \{0, 1, \dots, m\}$, G_k désigne le graphe $(S_n, (a_0, a_1, \dots, a_{k-1}))$ obtenu en ne conservant que les k premières arêtes de G.

Q3 On suppose que G est un arbre. Montrer que pour tout $k \in \{0, 1, \ldots, m-1\}$, les extrémités de a_k appartiennent à deux composantes connexes différentes de G_k. En déduire que $m = n - 1$.

Q4 Montrer que les trois propriétés suivantes sont équivalentes :

1. G est un arbre ;
2. G est connexe et $m = n - 1$.
3. G est acyclique et $m = n - 1$.

Nous souhaitons écrire une fonction qui teste si un graphe est un arbre. Nous allons pour cela utiliser une structure de données permettant de manipuler les partitions de l'ensemble S_n : si $\mathcal{P} = \{X_1, X_2, \ldots, X_k\}$ est une partition de S_n, on choisit dans chaque partie X_i, un élément particulier r_i, appelé représentant de X_i. Notre structure de données doit nous permettre :

- de calculer, pour $s \in S_n$, le représentant de la partie X_i contenant s ; cet élément sera également appelé représentant de s ;
- pour deux entiers s et t représentant des parties distinctes X_i et X_j, de transformer $\mathcal{P}$ en réunissant X_i et X_j, s ou t devenant le représentant de la partie $X_i \cup X_j$.

Nous représenterons une partition $\mathcal{P} = \{X_1, X_2, \ldots, X_k\}$ de S_n par une forêt : chaque X_i est représenté par un arbre dont les noeuds sont étiquetés par les éléments de X_i et de racine le représentant r_i de X_i , les arcs étant orientés vers la racine. Nous noterons $h(r_i)$ la hauteur de l'arbre X_i , c'est-à-dire la longueur de sa plus longue branche.

Ainsi, $\mathcal{P}_9 =$ `{{0, 2}, {1, 5, 6, 8}, {3, 4, 7}}` est une partition de S_9 et peut par exemple être représentée par la forêt de la figure suivante :

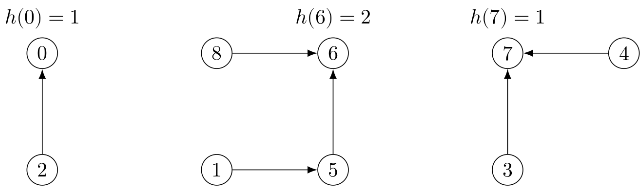

Figure 3 Une représentation de $\mathcal{P}_9 = \{\{0, 2\}, \{1, 5, 6, 8\}, \{3, 4, 7\}\}$

Le calcul du représentant d'un entier $s \in S_n$ se fait donc en remontant jusqu'à la racine de l'arbre (ce qui justifie l'orientation des arcs). Dans l'exemple, les représentants de 2, 1 et 7 sont respectivement 0, 6 et 7. Une partition $\mathcal{P}$ de $S_n = \{0, 1, \ldots, n-1\}$ sera représentée par un tableau d'entiers `P` de taille `n` de sorte que pour tout $s \in \{0, 1, \ldots, n-1\}$, $P[s]$ est le père de s si s n'est pas un représentant, et est égal à $-1 - h(s)$ si s est un représentant. La partition $\mathcal{P}_9$ peut ainsi être représentée par le tableau : `{-2, 5, 0, 7, 7, 6, -3, -2, 6}`

La réunion de deux parties X_i et X_j de représentants s et t distincts se fait selon la méthode suivante :

- si $h(s) > h(t)$, s est choisi pour représentant de la partie $X_i \cup X_j$ et devient le père de t ;
- si $h(s) \leq h(t)$, t est choisi pour représentant de la partie $X_i \cup X_j$ et devient le père de s.

Q5 Donner le résultat de la réunion de X_1 et X_2 dans $\mathcal{P}_9$.

Q6 Écrire une fonction `int representant(int* tab, int size, int s)` qui, appliquée à un tableau représentant une partition de S_n de taille `size` et à $s \in S_n$, renvoie le représentant de s. On demande que la fonction proposée soit récursive.

Q7 Écrire une fonction `void union(int* tab, int size, int s, int t)` qui, appliquée à un tableau représentant une partition de S_n de taille `size` et à deux représentants s et t distincts, modifie la partition en réunissant les arbres associés à s et t, selon la méthode expliquée ci-dessus, sans oublier, si nécessaire, de modifier $h(s)$ ou $h(t)$.

On note $\mathcal{P}_n(0)$ la partition de S_n où toutes les parties sont des singletons.

Q8 Soit $\mathcal{P}$ une partition de S_n construite à partir de $\mathcal{P}_n(0)$ par des réunions successives selon la méthode précédente. Montrer que si s est le représentant d'une partie $X \in \mathcal{P}$, alors le cardinal de X vérifie $|X| \geq 2^{h(s)}$.

Q9 En déduire les complexités des deux fonctions précédentes dans le pire des cas en fonction de n pour une partition $\mathcal{P}$ construite à partir de $\mathcal{P}_n(0)$.

Q10 Citer et expliquer le principe de la méthode au programme qui permet d'améliorer la complexité amortie de ces opérations. Ecrire une fonction `representant_opt` qui inclut cette optimisation.

Q11 Écrire une fonction `bool est_un_arbre(graphe g)` qui, appliquée à un graphe `g` représentant un graphe G, renvoie `true` si G est un arbre et `false` sinon.

1.2 Algorithme de Wilson : arbre couvrant aléatoire

Soit $G = (S_n, A)$ un graphe connexe et soit un entier $r \in S_n$. Un arbre couvrant de G enraciné en r est un arbre $T = (S_n, B)$ tel que $B \subset A$ et dont r a été choisi pour racine.

On convient de représenter un tel arbre en suivant la même idée que pour représenter une partition dans la partie précédente : par un tableau tel que la case s du tableau contient l'indice du père de s dans l'arbre si s n'est pas lui-même la racine, et -1 sinon :

```
struct arbre = {int size; int* peres;};
typedef struct arbre arbre;
```

Dans cette partie, nous supposons que $G = (S_n, A)$ est un graphe connexe et r un sommet de ce graphe. Nous cherchons à construire un arbre couvrant aléatoire de G enraciné en r.

Nous allons pour cela faire évoluer dynamiquement un arbre T , représenté par un tableau parent de longueur n vérifiant :

- la case r de parent contient -1 ;
- si $s \in S_n$ n'est pas un sommet de T, alors la case s de parent contient -2;
- si $s \in S_n$ est un sommet de T différent de la racine, alors la case s de parent contient le père de s dans T.

La construction de l'arbre T se fait en suivant l'algorithme 1.

Algorithme 1 Algorithme de Wilson de création d'un arbre couvrant aléatoire

entrée : G graphe connexe, r sommet de G
sortie : arbre couvrant aléatoire de G enraciné en r

$T \leftarrow (\{r\}, \emptyset)$
pour tout sommet s de G **faire**
    si s n'est pas un sommet de T **alors**
        $c \leftarrow$ chemin-aléatoire(G, T, s)
        greffer(c, T)
    fin si
fin pour
renvoyer T

La greffe d'un chemin élémentaire qui termine par un sommet de T et dont c'est le seul sommet en commun avec T se fait de manière naturelle, en ajoutant ce chemin.

Le calcul d'un chemin aléatoire dans le graphe entre un sommet qui n'appartient pas à l'arbre et un sommet qui appartient à l'arbre se fait en suivant l'algorithme 2.

Algorithme 2 Algorithme chemin-aléatoire

entrée : G graphe connexe, T arbre sous-graphe de G, s sommet de G et pas de T
sortie : $c = (s_0 = s, s_1, \ldots, s_k)$ chemin aléatoire élémentaire de G partant de s dont le seul sommet dans T est s_k

$c \leftarrow (s)$
tant que le dernier sommet de c n'appartient pas à T **faire**
    on note $c = (s_0, s_1, \ldots, s_k)$
    soit u un voisin de s_k dans G (choisi aléatoirement et uniformément)
    si il existe $i \in [\![0, k]\!]$ tel que $u = s_i$ **alors**
        $c \leftarrow (s_0, \ldots, s_i)$
    sinon $c \leftarrow (s_0, \ldots, s_k, u)$
    fin si
fin tant que
renvoyer c

On représente un chemin élémentaire en C par le type :

```
struct chemin = {int debut; int fin; int* suivant; int size;};
typedef struct chemin chemin;
```

de telle sorte que si le chemin $c = (s_0, s_1, \ldots, s_k)$ est représenté par c, alors :

- le champ `debut` de `c` contient s_0,
- son champ `fin` contient s_k,
- son champ `suivant` est un tableau de taille `size` et pour tout $j \in \{0, \ldots, k-1\}$, la case d'indice s_j de ce tableau contient la valeur s_{j+1},
- les valeurs des cases du champ `suivant` dont les indices n'appartiennent pas à $\{s_0, \ldots, s_{k-1}\}$ n'ont pas d'importance.

Q12 Expliquer pourquoi le type `chemin` ne peut pas représenter un chemin non élémentaire.

Q13 Écrire une fonction `void greffer(arbre* a, chemin* c)` qui implémente la greffe d'un chemin sur un arbre, en supposant (sans avoir à le vérifier) que le chemin possède un unique sommet qui appartient à l'arbre : son dernier sommet.

Q14 Écrire une fonction `chemin* chemin_aleatoire(graphe g,arbre* a,int s)` qui exécute l'algorithme 2. On pourra utiliser la commande `rand()%n` quirenvoie uniformément un entier compris entre 0 et $n-1$.

Q15 Écrire une fonction `arbre* wilson(graphe g,int r)` qui implémente l'algorithme 1. On prendra les initiatives nécessaires pour éviter les fuites mémoire tout en respectant les prototypes des fonctions imposées.

2 Problème sur les automates (Ocaml)

Langages et mots

On appelle *alphabet* tout ensemble fini de lettres. On note généralement l'alphabet Σ.

On note $\Sigma^\star$ l'ensemble de tous les mots formés sur l'alphabet Σ.

La *longueur* (ou la *taille*) d'un mot $w \in \Sigma^\star$ est son nombre de lettres et se note $|w|$. Le *mot vide*, noté ε, est le seul mot de longueur nulle.

Si un mot $w \in \Sigma^\star$ est de longueur $|w| = n$, on le note $w = a_0 a_1 ... a_{n-1}$, où les a_i sont des lettres de Σ.

Un *langage* sur l'alphabet Σ est un ensemble $L \subset \Sigma^\star$.

L'*étoile de Kleene* d'un langage L, notée $L^\star$, est le plus petit langage qui inclut L, qui contient ε et qui est stable par concaténation.

La concaténation de deux langages L et L' est notée $L \cdot L'$, souvent abrégé en LL' lorsqu'il n'y a pas d'ambiguïté.

Automates finis

Un *automate fini non déterministe* sur un alphabet Σ est un quadruplet $A = (Q, I, F, T)$, où Q est un ensemble fini d'*états*, $I \subset Q$ est le sous-ensemble des *états initiaux*, $F \subset Q$ est le sous-ensemble des *états finaux* et l'ensemble $T \subset Q \times \Sigma \times Q$ est l'ensemble des *transitions*, étiquetées par les lettres de l'alphabet Σ.

Si $(q, a, q') \in T$, on note $q \overset{a}{\to} q'$ cette transition.

Pour représenter graphiquement un automate, on utilise une flèche entrante pour désigner un état initial et une flèche sortante pour désigner un état final, comme l'illustre l'exemple de la figure 1.

Un mot $w = a_0 ... a_{n-1}$ est reconnu par l'automate A s'il existe une succession de transitions :

$$q_0 \overset{a_0}{\to} q_1 \overset{a_1}{\to} \cdots q_{n-1} \overset{a_{n-1}}{\to} q_n \qquad \text{avec} \qquad q_0 \in I \quad \text{et} \quad q_n \in F.$$

On dira que le mot w *étiquette un chemin* dans l'automate A allant de q_0 à q_n.

Le langage d'un automate A, noté L_A, est exactement l'ensemble des mots reconnus par l'automate A. On dit alors que A *reconnait* L_A. Un langage est dit *reconnaissable* s'il est le langage d'un automate fini.

Un *automate fini déterministe* sur un alphabet Σ est un quadruplet $A = (Q, \{q_0\}, F, \delta)$, où l'ensemble des états initiaux est un singleton (un unique état initial) et où l'ensemble des transitions T est remplacé par une fonction de transition δ définie sur un sous-ensemble de $Q \times \Sigma$ et à valeurs dans Q. Pour chaque couple $(q, a) \in Q \times \Sigma$, il existe au plus une transition (q, a, q') qui, si elle existe, est telle que $q' = \delta(q, a)$.

L'automate est déterministe *complet* si la fonction de transition δ est définie sur $Q \times \Sigma$. Dans ce cas, on définit la *fonction de transition étendue* $\delta^\star$ sur $Q \times \Sigma^\star$ par

$$\forall q \in Q, \quad \begin{cases} \delta^\star(q, \varepsilon) = q \\ \delta^\star(q, wa) = \delta\big(\delta^\star(q, w), a\big) \quad \forall w \in \Sigma^\star, \forall a \in \Sigma \end{cases}$$

Les automates seront représentés par le type Caml suivant

```
type automate = { nb : int;                                  (* nombre d'états *)
                  init : int list ;                          (* états initiaux *)
                  final : int list;                          (* états finaux *)
                  trans : (int * char * int) list} ;;        (* transitions *)
```

l'ensemble d'états Q d'un automate implémenté étant toujours supposé être un intervalle d'entiers $[\![0, n-1]\!]$.

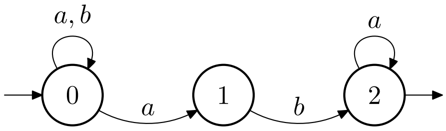

Figure 1 L'automate $\mathcal{A}_1$

Par exemple, l'automate $\mathcal{A}_1$ de la figure 1 est codé par

```
let a1 = { nb = 3 ;
           init = [0];
           final = [2];
           trans = [(0, 'a', 0); (0, 'a', 1); (0, 'b', 0); (1, 'b', 2); (2, 'a', 2)] } ;;
```

On accède au nombre d'états par `a1.nb`, à la liste des états initiaux par `a1.init`, à la liste des états finaux par `a1.final` et à la liste des transitions par `a1.trans`.

Expressions rationnelles

Soit Σ un alphabet. On définit la *syntaxe des expressions rationnelles* par :

— $\varnothing$, ε et a sont des expressions rationnelles, pour toute lettre $a \in \Sigma$;

— si E et F sont deux expressions rationnelles, alors $(E+F)$, $(E \cdot F)$ et $E^\star$ sont des expressions rationnelles.

La *sémantique des expressions rationnelles* est définie par l'application $\mathcal{L}$ qui associe à toute expression rationnelle un langage rationnel sur Σ par :

$$\begin{cases} \mathcal{L}(\varnothing) = \emptyset & \text{(langage vide)} \\ \mathcal{L}(\varepsilon) = \{\varepsilon\} & \text{(langage contenant le mot vide)} \\ \mathcal{L}(a) = \{a\} & \forall a \in \Sigma \end{cases}$$

et, si E et F sont deux expressions rationnelles,

$$\begin{cases} \mathcal{L}(E+F) = \mathcal{L}(E) \cup \mathcal{L}(F) \\ \mathcal{L}(E \cdot F) = \mathcal{L}(E) \cdot \mathcal{L}(F) \\ \mathcal{L}(E^\star) = \mathcal{L}(E)^\star \end{cases}$$

où $\star$ représente l'étoile de Kleene d'un langage et $\cdot$ représente la concaténation de deux langages.

Programmation

Le seul langage de programmation autorisé dans cette épreuve est Caml. Toutes les fonctions des modules `Array` et `List`, ainsi que les fonctions de la bibliothèque standard (celles qui s'écrivent sans nom de module, comme `max` ou `incr` ainsi que les opérateurs comme `/` ou `mod`) peuvent être librement utilisés.

Généralement, les objets mathématiques dans le texte seront notés A, m, i, n, ℓ, alors qu'ils seront représentés en Caml par `a`, `m`, `i`, `n`, `l`.

Les complexités demandées sont des complexités temporelles dans le pire des cas et seront exprimées sous la forme $O\big(f(n,m)\big)$, où f est une fonction usuelle simple et où n et m sont des paramètres correspondant aux tailles des objets en entrée de l'algorithme.

I Mots et automates

I.A – Miroir d'un mot et automate transposé

Pour tout mot $w = a_0 a_1 ... a_{n-1}$ de longueur $n \in \mathbb{N}^*$, on définit son *mot miroir* $\tilde{w}$ par $\tilde{w} = a_{n-1} ... a_1 a_0$. Par convention, le mot vide ε est son propre miroir.

Pour tout langage $L \subset \Sigma^\star$, on définit son *langage miroir* $\tilde{L}$ constitué de l'ensemble des mots miroirs du langage L :

$$\tilde{L} = \{\tilde{w} \mid w \in L\}.$$

Q 1. Décrire le langage L_1 de l'automate $\mathcal{A}_1$ de l'exemple de la figure 1 et décrire son langage miroir $\tilde{L}_1$.

Q 2. Dessiner un automate $\tilde{\mathcal{A}}_1$, reconnaissant le langage miroir $\tilde{L}_1$.

Soit $A = (Q, I, F, T)$ un automate non déterministe et $L = L_A$ le langage qu'il reconnait.

Q 3. Donner, en justifiant, la construction de l'automate miroir $\tilde{A} = (Q, I', F', T')$ qui reconnait le langage $\tilde{L}$.

Q 4. Écrire une fonction `transpose` de signature `automate -> automate` qui étant donné un automate A non déterministe en entrée, renvoie un automate non déterministe qui reconnait le miroir de L_A.

Q 5. Quelle est la complexité de cette fonction ?

I.B – Palindromes et rationalité

Soit $w \in \Sigma^\star$. On dit que le mot w est un *palindrome* si $\tilde{w} = w$.

Q 6. Écrire une fonction `palindrome` de signature `string -> bool` qui teste, en temps linéaire, si un mot est un palindrome.

On rappelle que pour tout $0 \leqslant i <$ (`String.length s`), le i-ième caractère de la chaine de caractères `s` est obtenu par l'expression `s.[i]`.

Pour un alphabet Σ, on note $\mathrm{Pal}(\Sigma)$ l'ensemble des palindromes de $\Sigma^\star$.

Q 7. Montrer que si Σ est un alphabet à une lettre, alors $\mathrm{Pal}(\Sigma)$ est rationnel.

Q 8. Montrer que si Σ contient au moins deux lettres, alors $\mathrm{Pal}(\Sigma)$ n'est pas rationnel.

On pourra utiliser un automate et un mot de $\mathrm{Pal}(\Sigma) \cap a^\star ba^\star$.

Soit $L \subset \Sigma^\star$ un langage reconnu par l'automate $A = (Q, I, F, T)$.

Pour $(q, q') \in Q^2$, on note $L_{q,q'}$ le langage de tous les mots w qui étiquettent un chemin dans A partant de q et arrivant en q'.

Q 9. Montrer que $L_{q,q'}$ est reconnaissable et exprimer le langage L_A en fonction de langages $L_{q,q'}$.

Q 10. Montrer que $\mathrm{Pal}(\Sigma) \cap (\Sigma^2)^\star = \{u\tilde{u} \mid u \in \Sigma^\star\}$.

Soit L un langage rationnel reconnu par un automate $A = (Q, I, F, T)$.

On définit les langages $D(L) = \{w\tilde{w} \mid w \in L\}$ et $R(L) = \{w \in \Sigma^\star \mid w\tilde{w} \in L\}$.

Q 11. Décrire simplement les langages $D(a^\star b)$ et $R(a^\star b^\star a^\star)$.

Q 12. Les langages $D(L)$ et $R(L)$ sont-ils reconnaissables ?

On pourra faire intervenir les langages $L_{q,q'}$, définis ci-dessus.

I.C – Déterminisation

On rappelle que pour tout automate $A = (Q, I, F, T)$ non déterministe, on peut définir l'automate déterminisé accessible $A_{\mathrm{det}} = (Y, \{I\}, F', \delta)$ où $Y \subset \mathcal{P}(Q)$ est l'ensemble des états accessibles depuis l'état initial $\{I\}$ dans l'automate des parties. Cet automate déterminisé accessible reconnait le même langage que l'automate A.

Q 13. Écrire un automate $\mathcal{A}_2$ non déterministe à 4 états qui reconnait le langage $L_2 = (b + ab)^\star ba$. Cet automate devra avoir un unique état initial et un unique état final.

Q 14. Appliquer l'algorithme de déterminisation sur l'automate miroir $\widetilde{\mathcal{A}_2}$ afin d'obtenir l'automate $\mathcal{A}_3 = \left(\widetilde{\mathcal{A}_2}\right)_{\mathrm{det}}$. Les états de $\mathcal{A}_3$ seront renommés $e_0, e_1, \ldots$.

Q 15. Appliquer l'algorithme de déterminisation sur l'automate miroir $\widetilde{\mathcal{A}_3}$ afin d'obtenir l'automate $\mathcal{A}_4 = \left(\widetilde{\mathcal{A}_3}\right)_{\mathrm{det}}$. Ses états seront renommés $q_0, q_1, \ldots$.

Q 16. Quel doit être le langage reconnu par l'automate $\mathcal{A}_4$?

On cherche à généraliser cette construction de façon effective. Pour cela, on va implémenter l'algorithme de déterminisation.

Il faut d'abord choisir une représentation pour les parties de Q (c'est-à-dire des ensembles d'états). Une solution naïve consisterait à utiliser des listes d'états. Lors du déroulement de l'algorithme de déterminisation, on peut être amené à effectuer des réunions d'ensembles. Une concaténation simple des listes génère des doublons qu'il faut ensuite supprimer afin que les listes codent bien des ensembles d'états.

Q 17. Écrire une fonction `supprimer` de signature `'a list -> 'a list` qui prend une liste en entrée et supprime toutes les occurrences multiples de ses éléments.

Q 18. Donner la complexité de votre algorithme en fonction de la taille de la liste d'entrée.

On choisit plutôt de coder les ensembles d'états par des entiers.

Pour un automate $A = (Q, I, F, T)$ tel que $Q = [\![0, n-1]\!]$, toute partie de Q va être représentée par un entier entre 0 et $2^n - 1$. Dans la suite, on supposera $n \leqslant 20$. Soit X une partie de $[\![0, n-1]\!]$. On définit le numéro de X par la fonction suivante

$$\mathrm{numero}(X) = \sum_{i \in X} 2^i.$$

On se donne `pow` un tableau des puissances de 2, qui contient toutes les puissances 2^k, pour $0 \leqslant k \leqslant 20$.

```
let pow = Array.make 21 1 ;;
  for i = 1 to 20 do
    pow.(i) <- pow.(i-1) * 2
  done ;;
```

Soit $q \in [\![0, n-1]\!]$ un état et $k \in [\![0, 2^n - 1]\!]$ le numéro d'un ensemble d'états X, c'est-à-dire $\mathrm{numero}(X) = k$.

Q 19. Écrire une fonction `est_dans` de signature `int -> int -> bool` qui teste, à l'aide d'opérations arithmétiques, si l'état q est dans l'ensemble d'états représenté par le numéro k en $O(1)$ opérations.

Soit ℓ une liste d'états contenant éventuellement plusieurs fois le même état, représentant l'ensemble X.

Q 20. Écrire une fonction `numero` de signature `int list -> int` qui calcule le numéro de l'ensemble X.

Par exemple $\ell = [1; 5; 2; 5; 2; 5; 2; 2; 1; 2; 1]$ représente l'ensemble $X = \{1, 2, 5\}$, de numéro $38 = 2^1 + 2^2 + 2^5$.

Soit ℓ une liste d'états et X un ensemble d'états représenté par son numéro k.

Q 21. Écrire une fonction `intersecte` de signature `int list -> int -> bool` qui vérifie si un élément de ℓ est contenu dans l'ensemble X représenté par k.

On prépare désormais la fonction de transition de l'automate déterminisé accessible.

Soit X un ensemble d'états de Q. On suppose désormais que l'automate est sur l'alphabet à deux lettres $\Sigma = \{a, b\}$.

On cherche à calculer la fonction de transition $\delta : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$ de l'automate déterminisé. On rappelle que, pour $c \in \{a, b\}$ et $X \in \mathcal{P}(Q)$,

$$\delta(X, c) = \bigcup_{q \in X} \{q' \in Q \mid (q, c, q') \in T\}.$$

La transition $(X, c, \delta(X, c))$ sera alors dans l'automate déterminisé.

En parcourant l'ensemble des transitions T de l'automate, on va simultanément calculer les états $(\delta(X, a), \delta(X, b))$, ce qui correspond à la table de transition depuis l'état X.

Q 22. Écrire une fonction `etat_suivant` de signature `int -> (int*char*int) list -> (int*int)` qui, étant donné en entrée un entier k tel que $k = \text{numero}(X)$ et la liste des transitions T, calcule le couple d'entiers (k_a, k_b) tels que $k_a = \text{numero}(\delta(X, a))$ et $k_b = \text{numero}(\delta(X, b))$.

Au moment de construire l'automate déterminisé accessible A_{det}, on va être amené à renommer (c'est-à-dire ici renuméroter) les états de A_{det} pour avoir au final un ensemble d'états Y de la forme $[\![0, N-1]\!]$ où N sera le nombre de parties de Q accessibles dans l'automate des parties. Pour cela, on va simplement utiliser une liste contenant des couples (k, v) où k est le numéro d'un ensemble d'états X et v le numéro final par lequel k sera remplacé. Par exemple, si à un moment donné de l'algorithme, la liste contient $(6, 2)$, "l'ensemble d'états 6" (qui correspond dans $\mathcal{P}(Q)$ à $\{1, 2\}$) est renuméroté 2.

Q 23. Écrire une fonction `cherche` de signature `int -> (int*int) list -> int` qui renvoie le nouveau numéro d'un ensemble d'états représenté par son numéro k dans une liste comme ci-dessus (-1 si k n'est pas présent).

Q 24. Écrire une fonction `determinise` de signature `automate -> automate` qui calcule le déterminisé accessible de l'automate d'entrée. On expliquera brièvement la démarche utilisée.

Q 25. Quelle est la complexité de votre fonction `determinise` en fonction du nombre d'états n de A et du nombre d'états N de A_{det} ?

I.D – Algorithme de Brzozowski

L'algorithme de Brzozowski permet d'obtenir un automate déterministe ayant un nombre minimal d'états, reconnaissant le même langage que l'automate initial.

On se donne un automate $A = (Q, I, \{f\}, T)$ qui reconnait le langage L et tel que l'automate miroir $\tilde{A}$ est déterministe et accessible.

On note $A_{\text{det}} = (Y, \{I\}, F, \delta)$ le déterminisé accessible de A.

Si u est un mot et L un langage, on note $u^{-1}L = \{w \in \Sigma^\star \mid uw \in L\}$.

Q 26. Soit $q \in Q$ un état et $u \in \Sigma^\star$ un mot. Montrer que si $q \in \delta^\star(\{I\}, u)$, alors il existe un mot $w \in \Sigma^\star$ tel que $uw \in L$.

Q 27. Montrer la propriété $(*)$: si l'on prend deux mots u et v dans $\Sigma^\star$ tels que $u^{-1}L = v^{-1}L$, alors dans l'automate A_{det} déterminisé, $\delta^\star(\{I\}, u) = \delta^\star(\{I\}, v)$.

Q 28. En déduire que si A est un automate quelconque reconnaissant L, alors en posant $B = \left(\tilde{A}\right)_{\text{det}}$, montrer que $\left(\tilde{B}\right)_{\text{det}}$ reconnait L et vérifie la propriété $(*)$.

Q 29. Écrire une fonction `minimal` de signature `automate -> automate` appliquant la construction de Brzozowski sur l'automate d'entrée. On fera abstraction de la taille des automates générés, possiblement problématique.

