Partie I - Coloration de graphes

Question 1 – Soit S_1 l'ensemble des sommets de G_1 et $c : S_1 \to \{0, 1\}$ telle que :

$$c(0) = 0 \qquad \text{et} \qquad c(1) = c(2) = c(3) = 1.$$

Alors c est une 2-coloration de G_1. Ainsi :

$$\boxed{\text{Le graphe } G_1 \text{ est 2-coloriable.}}$$

Question 2 – Montrons que :

$$\boxed{\text{Pour tout } n \geqslant 1, \text{ le nombre chromatique de } K_n \text{ est } n.}$$

Soit $n \geqslant 1$ et $S = \{x_0, x_1, \ldots, x_{n-1}\}$ l'ensemble des sommets de K_n. Alors la fonction :

$$c : S \to \{0, 1, \ldots n-1\} \qquad \text{telle que} \qquad \forall i \in \{0, 1, \ldots n-1\} : c(x_i) = i$$

est une coloration de K_n. Ainsi :

$$\chi(K_n) \leqslant n.$$

Montrons maintenant que $\chi(K_n) \geqslant n$. Soit $k \in \mathbb{N}$ tel que $k < n$ et $c : S \to \{0, 1, \ldots k-1\}$ une fonction quelconque. Comme $k < n$, la fonction c n'est pas injective, c'est à dire qu'il existe deux sommets $(x, y) \in S^2$ tels que $x \neq y$ et $c(x) = c(y)$. Étant donné que K_n est complet, $\{x, y\}$ est une arête de G et donc c n'est pas une coloration. On en déduit que :

$$\text{Si } k < n, \text{ alors aucune fonction } c : S \to \{0, 1, \ldots k-1\} \text{ n'est une coloration de } K_n.$$

Ainsi, $\chi(K_n) \geqslant n$ d'où le résultat.

Question 3 – Il s'agit d'une généralisation du résultat de la question précédente, la preuve est similaire. Soit $n \geqslant 1$ et $S = \{x_0, x_1, \ldots, x_{n-1}\}$ l'ensemble des sommets de G. Alors la fonction :

$$c : S \to \{0, 1, \ldots n-1\} \qquad \text{telle que} \qquad \forall i \in \{0, 1, \ldots n-1\} : c(x_i) = i$$

est une coloration de G. Ainsi :

$$\boxed{\chi(G) \leqslant n.}$$

Montrons maintenant que $\chi(G) \geqslant \omega(G)$. Soit $k \in \mathbb{N}$ tel que $k < \omega(G)$ et $c : S \to \{0, 1, \ldots k-1\}$ une fonction quelconque. Par définition de $\omega(G)$, il existe un ensemble de sommet $S' \subset S$ tel que $\text{card}(S') = \omega(G)$ et dont les éléments sont adjacents 2 à 2. Soit $c' : S' \to \{0, 1, \ldots k-1\}$ la restriction de c à S'. Comme $k < \omega(G)$, la fonction c' n'est pas injective, c'est à dire qu'il existe deux sommets $(x, y) \in S'^2$ tels que $x \neq y$ et $c'(x) = c'(y)$. Étant donné que S' est une clique, $\{x, y\}$ est une arête de G et donc c n'est pas une coloration. On en déduit que :

$$\text{Si } k < \omega(G), \text{ alors aucune fonction } c : S \to \{0, 1, \ldots k-1\} \text{ n'est une coloration de } G.$$

En d'autres termes :

$$\boxed{\chi(K_n) \geqslant \omega(G).}$$

Question 4 – On obtient :

```
d1 = { 0: [1,2,3], 1: [0], 2: [0], 3: [0] }
```

Question 5 –

```
# Solution 1
def degres_sommets(d):
    L = []
    for i in d:
        L.append((len(d[i]), i))
    return L
```

```
# Solution 2
def degres_sommets(d):
    return [(len(d[i]), i) for i in d]
```

Question 6 –

```
# La question 6 évoque une fonction 'tri'. En voici une implémentation:
def tri(L):
    return sorted(L, reverse = True)
```

```
def tri_degres(d):
    L = tri(degres_sommets(d))
    return [s for (_,s) in L]
```

Question 7 –

```
# La fonction test1 vérifie que les valeurs de c appartiennent toutes à {0,1}.
def test1(c):
    for x in c:
        if c[x] != 0 and c[x] != 1:
            return False
    return True
```

```
# La fonction test2 vérifie que l'ensemble des clés de d et l'ensemble des clés
# de c sont identiques.
# Remarque: j'imagine qu'une réponse sans cette fonction aurait été acceptée le
# jour de l'épreuve.
def test2(d, c):
    if len(d) != len(c):
        return False
    for x in d:
        if not (x in c):
            return False
    return True
```

```
# La fonction test3 vérifie que pour toute arête {x,y}, les sommets x et y n'ont
# pas la même couleur.
# Hypothèse: test2(d,c) vaut True.
def test3(d, c):
    for x in d:
        for y in d[x]:
            if c[x] == c[y]:
                return False
    return True
```

```
def test(d, c):
    return test1(c) and test2(d, c) and test3(d, c)
```

Question 8 – Il semble y avoir une erreur d'énoncé : dans l'algorithme 1, la ligne commençant par "**Sortie**" indique que `colorie` est une liste alors que la ligne 3 indique que c'est un dictionnaire. Dans la suite, on considère que c'est un dictionnaire.

Extrait du rapport du jury : "Compte tenu de l'énoncé de l'algorithme, les 2 types de retour liste et dictionnaire ont été acceptés dans les implémentations proposées."

Lorsqu'on ordonne les sommets de G_1 selon les degrés décroissants (à l'aide de la fonction `tri_degres` de la question 6), on obtient `[0, 3, 2, 1]`.

Avant d'entrer dans la boucle **Tant que**, le dictionnaire `colorie` est vide. À la fin du premier tour de boucle, il vaut `{0: 0}`. À la fin du deuxième tour de boucle, il vaut `{0: 0, 3: 1, 2: 1, 1: 1}` et l'algorithme s'arrête. Finalement :

Si on déroule l'algorithme avec G_1 en entrée, on obtient `{0: 0, 3: 1, 2: 1, 1: 1}`

Question 9 –

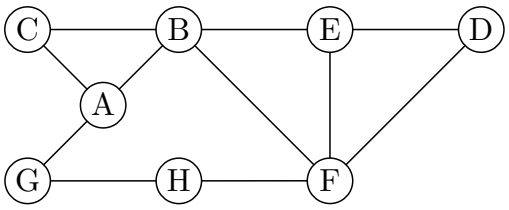
```
def adjacent(d, dc, s, c):
    for x in d[s]:
        if x in dc and dc[x] == c:
            return True
    return False
```

Question 10 –

```
def Q10_welsh_powel(d):
    li = tri_degres(d)
    colorie = {}
    c = 0
    while len(colorie) < len(d):
        for s in li:
            if not (s in colorie) and not adjacent(d, colorie, s, c):
                colorie[s] = c
        c += 1
    return colorie
```

Remarque. Extrait du rapport du jury concernant la question 10 : "Compte tenu de l'énoncé de l'algorithme, les 2 types de retour liste et dictionnaire ont été acceptés dans les implémentations proposées."

Question 11 – On modélise cette situation par un graphe dont les sommets sont les étudiants et les arêtes représentent les liens d'amitié (deux étudiants x et y sont adjacents dans le graphe si et seulement si ils sont amis). On obtient :



On cherche alors une coloration de ce graphe. Chaque couleur va représenter un groupe de travail et la définition de "coloration" permet de garantir que chaque étudiant est dans un groupe de différent de celui de ses amis. Par exemple, si on applique l'algorithme de Welsh-Powel, on obtient la coloration :

```
{"F": 0, "A": 0, "B": 1, "H": 1, "D": 1, "E": 2, "G": 2, "C": 2}
```

Ainsi :

Une solution au problème consiste à créer les groupes de travail $\{A, F\}$, $\{B, D, H\}$ et $\{E, C, G\}$.

Partie II - Satisfiabilité d'une formule propositionnelle

```
(* Le type suivant est copié-collé depuis la page 4 de l'énoncé *)
type clause = Var of int | Non of clause | Ou of clause * clause;;
```

Question 12 –

```
let c_Q12 = Ou (Ou (Var 0, Var 1), Non (Var 2))
```

Question 13 –

```
let f_Q13 = [c_Q12; Ou (Non (Var 1), Var 2)];;
```

Question 14 –

```
let rec evalue_clause (c: clause) (v: bool array): bool = match c with
  | Var i -> v.(i)
  | Non (c1) -> not (evalue_clause c1 v)
  | Ou (c1, c2) -> evalue_clause c1 v || evalue_clause c2 v;;
```

Question 15 –

```
let rec evalue_FNC (f: clause list) (v: bool array): bool = match f with
  | [] -> true
  | c :: f1 -> evalue_clause c v && evalue_FNC f1 v;;
```

Question 16 – Avec la formule $F = (x_0 \vee x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$ et le tableau de valuations `[|false; true; true|]` :

On obtient `true`

En effet :

→ La première clause s'évalue en `true` car elle contient le littéral x_1 dont la valuation est `true`.

→ La deuxième clause s'évalue en `true` car elle contient le littéral x_2 dont la valuation est `true`.

Question 17 –

```
(* Attention: cette fonction modifie v (même si la valeur renvoyée est false) *)
let suivant (v: bool array): bool =
  let i = ref (Array.length v - 1) in
  while !i >= 0 && v.(!i) do
    v.(!i) <- false;
    i := !i - 1
  done;
  match !i with
  | -1 -> false
  | _ -> v.(!i) <- true;
         true;;
```

Question 18 –

```
let satisfiable (f: clause list) (n: int): bool =
  let v = Array.make n false in
  let res = ref (evalue_FNC f v) in
  while not !res && suivant v do
    res := evalue_FNC f v
  done;
  !res;;
```

Question 19 – Comme dans l'énoncé, notons n le nombre de variables propositionnelles. Étant données une clause c et F une formule sous forme normale conjonctive dont $c_0, \dots c_{m-1}$ sont les clauses :

$$F = \bigwedge_{k=0}^{m-1} c_k,$$

on note $\ell(c)$ la taille de c (c'est à dire le nombre de noeuds dans l'arbre représentant c) et $\ell(F)$ la somme des tailles des clauses de F :

$$\ell(F) = \sum_{k=0}^{m-1} \ell(c_k).$$

On remarque alors que :

→ Le temps d'exécution de `evalue_clause` est en $\mathcal{O}(\ell(c))$ où c est la clause en entrée.

→ Le temps d'exécution de `evalue_FNC` avec une formule F en entrée est :

$$\mathcal{O}\left(\sum_{k=0}^{m-1} \ell(c_k)\right) = \mathcal{O}(\ell(F))$$

→ Le temps d'exécution de `suivant` est en $\mathcal{O}(n)$.

→ Dans la fonction `satisfiable`, l'appel à `Array.make` se fait en temps $\mathcal{O}(n)$, il y a $\mathcal{O}(2^n)$ appels à la fonction `evalue_FNC` et $\mathcal{O}(2^n)$ appels à la fonction `suivant`.

Finalement :

Le temps d'exécution de `satifiable` est en $\mathcal{O}\Big(2^n(n+\ell(F))\Big)$ où $\ell(F)$ est la taille de F (c'est à dire le nombre total de noeuds dans les arbres représentant les clauses de F)

Question 20 – On utilise une procédure récursive qui prend en entrée un entier $i \in [\![0, n]\!]$, qui renvoie `true` si la formule est satisfiable et `false` sinon.

Cas de base. Lorsque $i = n$, cela signifie que la valuation de toutes les variables $x_0, \dots x_{n-1}$ a été fixée. On renvoie alors le résultat de l'appel à la fonction `evalue_FNC`.

Cas général. Lorsque $i \in [\![0, n-1]\!]$, cela signifie que la valuation des variables $x_0, \dots x_{i-1}$ a été fixée, mais pas celle de $x_i, \dots x_{n-1}$. On commence par fixer la valuation de x_i à `false` et on lance un appel récursif sur $i+1$. Si cet appel récursif renvoie `true`, c'est que la formule est satisfiable, on peut donc renvoyer `true`. Sinon, on fixe la valuation de x_i à `true` et on renvoie le résultat de l'appel récursif sur $i+1$.

Remarque. Il semble y avoir une erreur d'énoncé dans la définition de $F \vDash \phi$ juste avant la question 21 : la phrase "tout modèle de ϕ est un modèle de Γ" est à remplacer par "tout modèle de Γ est un modèle de ϕ".

Question 21 – On remarque que l'assertion "F est une conséquence logique d'un ensemble de formules $F_1, \dots, F_n$" est équivalent à dire que $F_1 \wedge \dots \wedge F_n \wedge \neg F$ n'est pas satisfiable. On en déduit un algorithme en pseudo-code :

→ On construit une formule φ sous forme normale conjonctive équivalente à $F_1 \wedge \dots \wedge F_n \wedge \neg F$.

→ On teste si φ est satisfiable à l'aide de la fonction précédente.

→ Si φ est satisfiable, on renvoie `false`, sinon on renvoie `true`.

Question 22 – Cette méthode se justifie par les théorèmes de correction et de complétude de la logique propositionnelle. Ces théorèmes stipulent que pour tout ensemble de formules Γ et toute formule F :

$$\Gamma \vdash \phi \Rightarrow \Gamma \vDash \phi \qquad \text{(théorème de correction)}$$
$$\Gamma \vDash \phi \Rightarrow \Gamma \vdash \phi \qquad \text{(théorème de complétude)}$$

Prouvons le séquent $\Gamma \vdash P$ où $\Gamma = \{P \to Q, Q \to R, P\}$ en en donnant un arbre de preuve utilisant la règle de l'axiome (ax) :

$$\frac{}{\Gamma \vdash P}\ \text{ax}$$

Remarque. Il y a certainement une erreur d'énoncé : le séquent à prouver est plutôt $\Gamma \vdash R$. Montrons ce séquent à l'aide de la règle d'élimination de l'implication ($\to_e$) et de la règle de l'axiome (ax) :

$$\dfrac{\dfrac{}{\Gamma \vdash Q \to R}\,\text{ax} \qquad \dfrac{\dfrac{}{\Gamma \vdash P \to Q}\,\text{ax} \qquad \dfrac{}{\Gamma \vdash P}\,\text{ax}}{\Gamma \vdash Q}\,\to_e}{\Gamma \vdash R}\,\to_e$$

Partie III - Automates et reconnaissance de motifs

```
(* Le type suivant est copié-collé depuis la page 7 de l'énoncé *)
type auto = {
    etats : int list;
    alphabet : char list;
    initial : int;
    transition : int -> char -> int ;
    final : int list
  };;
```

Question 23 – Soit $x \in A^*$ un mot non vide, alors $p = |x|$ est nécessairement une période de x. En effet :

$$\begin{aligned} p = |x| \text{ est une période de } x &\Leftrightarrow \forall i \in [\![0; |x| - p - 1]\!] : x_i = x_{i+p} \\ &\Leftrightarrow \forall i \in \varnothing : x_i = x_{i+p} \end{aligned}$$

Question 24 –

```
(* occurrence_aux renvoie un booléen qui indique si x est un facteur de y à
   partir de l'indice i0, c'est à dire si x = y.[i0, ..., i0 + |x| - 1].
   Hypothèse: 0 <= i0 <= |y| - |x|*)
let occurrence_aux (x: string) (y: string) (i0: int): bool =
  let lx = String.length x in
  let i = ref 0 in
  while !i < lx && x.[!i] = y.[i0 + !i] do
    incr i;
  done;
  !i = lx;;
```

```
let occurrence (x: string) (y: string): bool =
  let lx = String.length x in
  let ly = String.length y in
  let i0 = ref 0 in
  while !i0 <= ly - lx && not (occurrence_aux x y !i0) do
    incr i0
  done;
  !i0 <= ly - lx;;
```

Question 25 – On remarque que le temps d'exécution de la fonction `occurence_aux` est en $\mathcal{O}(|x|)$ car la boucle `while` fait $\mathcal{O}(|x|)$ tours et toutes les opérations utilisées s'exécutent en temps constant.

Pour la fonction `occurence`, la boucle `while` fait $\mathcal{O}(|y|)$ tours et chaque tour s'exécute en temps $\mathcal{O}(|x|)$ à cause de l'appel à `occurence_aux`. Finalement :

La complexité en temps de la fonction `occurence` est en $\mathcal{O}(\|x\| \times \|y\|)$.

Question 26 – Pour $x = aabaabaa$:

→ $p = 1$ n'est pas une période car $x_1 = a \neq b = x_{1+1}$

→ $p = 2$ n'est pas une période car $x_0 = a \neq b = x_{0+2}$

→ $p = 3$ est une période.

Finalement :

La période de $x = aabaabaa$ est 3

Question 27 –

```
(* La fonction est_periode renvoie un booléen qui indique si p est une période
   de x *)
let est_periode (x: string) (p: int): bool =
  let lx = String.length x in
  let i = ref 0 in
  while !i <= lx - p - 1 && x.[!i] = x.[!i+p] do
    incr i
  done;
  !i = lx - p;;
```

```
(* Hypothèse: x est non vide *)
let periode (x: string): int =
  if String.length x = 0 then failwith "periode: x est vide";
  let p = ref 1 in
  while not (est_periode x !p) do
    incr p;
  done;
  !p;;
```

Question 28 – Montrons l'équivalence par double implication.

($\Rightarrow$) Supposons que p soit une période de x. Soient u, v, w les trois chaînes de caractères définies par :

$$u = x_0\ x_1\ \dots\ x_{p-1} \qquad v = x_{|x|-p}\ x_{|x|-p+1}\ \dots\ x_{|x|-1} \qquad w = x_p\ x_{p+1}\ \dots\ x_{|x|-1}$$

On a :

$$x = uw \qquad \begin{aligned}|u| &= p - 1 - 0 + 1\\ &= p\end{aligned} \qquad \begin{aligned}|v| &= |x| - 1 - (|x| - p) + 1\\ &= p\end{aligned} \qquad \begin{aligned}|w| &= |x| - 1 - p + 1\\ &= |x| - p\end{aligned}$$

Montrons que $x = wv$. Comme p est une période de x :

$$\begin{aligned}\forall i \in [\![0; |w| - 1]\!] : x_i &= x_{i+p}\\ &= (uw)_{i+p}\\ &= w_{i-|u|+p}\\ &= w_i\\ &= (wv)_i\end{aligned} \qquad \begin{aligned}\forall i \in [\![|w|; |x| - 1]\!] : x_i &= x_{|x|-p+(i-|x|+p)}\\ &= v_{i-|x|+p}\\ &= v_{i-|w|}\\ &= (wv)_i\end{aligned}$$

($\Leftarrow$) Supposons qu'il existe 3 chaînes de caractères u, v, w telles que $x = uw = wv$ et $|u| = |v| = p$. On a alors :

$$|w| = |x| - |u| = |x| - p$$

Montrons que p est une période de x :

$$\begin{aligned}\forall i \in [\![0; |x| - p - 1]\!] : x_i &= (wv)_i\\ &= w_i\\ &= (uw)_{i+|u|}\\ &= x_{i+|u|}\\ &= x_{i+p}\end{aligned}$$

Question 29 – Pour toute chaîne de caractères x non vide, notons $\mathcal{B}(x)$ l'ensemble des bords de x et $n(x)$ le plus grand entier tel que $Bord^{n(x)}(x)$ est bien défini. On remarque que :

$$\forall k \in \mathbb{N}^* : Bord^k(Bord(x)) = Bord^{k+1}(x)$$

- Dans un premier temps, montrons par récurrence sur $|x| \in \mathbb{N}^*$ que :

$$\mathcal{B}(x) = \Big\{ Bord^k(x) : k \in [\![1; n(x)]\!] \Big\}$$

Soit $k \geqslant 1$ tel que la propriété soit vraie pour tout mot x' vérifiant $|x'| < k$. Soit x un mot tel que $|x| = k$.

1er cas. Supposons que $n(x) = 1$. Alors $Bord(x) = \varepsilon$, c'est à dire que le seul bord de x est ε. Ainsi :

$$\mathcal{B}(x) = \{\varepsilon\} = \Big\{ Bord^k(x) : k \in [\![1; n(x)]\!] \Big\}$$

2ème cas. Supposons que $n(x) \geqslant 2$. Alors $Bord(x) \neq \varepsilon$ et il est clair que $n(Bord(x)) = n(x) - 1$. Ainsi, l'hypothèse de récurrence appliquée à $Bord(x)$ donne :

$$\begin{aligned}
\mathcal{B}(Bord(x)) &= \Big\{ Bord^k(Bord(x)) : k \in [\![1; n(Bord(x))]\!] \Big\} \\
&= \Big\{ Bord^{k+1}(x) : k \in [\![1; n(x) - 1]\!] \Big\} \\
&= \Big\{ Bord^k(x) : k \in [\![2; n(x)]\!] \Big\}
\end{aligned}$$

Pour conclure il suffit de remarquer que :

$$\mathcal{B}(x) = \mathcal{B}(Bord(x)) \cup \{Bord(x)\}$$

Conclusion. En conclusion, $Bord(x)$, $Bord^2(x)$, …, $Bord^n(x)$ est la suite des bords de x. Par ailleurs, il est clair que les mots de cette suite apparaissent par ordre décroissant de leur longueur.

- Soit x une chaîne de caractères non vide et $p \in [\![1; |x|]\!]$. D'après la question 28 :

$$\begin{aligned}
p \text{ est une période de } x &\Leftrightarrow \exists (u, v, w) : \Big[x = uw = wv \text{ et } |u| = |v| = p \Big] \\
&\Leftrightarrow \exists w : \Big[w \text{ est un préfixe de } x, w \text{ est un suffixe de } x \text{ et } |w| = |x| - p \Big] \\
&\Leftrightarrow \exists w \in \mathcal{B}(x) : |w| = |x| - p
\end{aligned}$$

Ainsi, le résultat de la première partie de la question implique que $|x| - |Bord(x)|$, $|x| - |Bord^2(x)|$, … , $|x| - |Bord^n(x)|$ est la suite des périodes de x dans l'ordre croissant.

Question 30 – Dans la fonction `occurence` de la question 24, on remarque qu'un caractère de `y` peut être vu jusqu'à `x` fois. On peut améliorer la procédure en évitant certaines comparaisons. L'idée est que si :

```
x.[0...i-1] = y.[i0...i0+i-1]
```

alors pour tout préfixe `z` de `x.[0...i-1]`, on a :

`z = y.[i0+i-|z|...i0+i-1]` ⇔ `z` est un bord de `x.[0...i-1]`.

Avec l'implémentation de la question 24, lorsqu'on se rend compte que `x.[!i] <> y.[i0+!i]` dans la fonction `occurence_aux`, alors on relance cette fonction avec :

$$\texttt{i0}' = \texttt{i0} + 1 \qquad \text{et} \qquad \texttt{i}' = 0$$

Grâce à l'équivalence ci-dessus, on remarque que si $\texttt{i} \neq 0$, alors on peut se contenter de relancer la fonction avec :

$$\texttt{i0}' = \texttt{i0} + \texttt{i} \qquad \text{et} \qquad \texttt{i}' = |Bord(\texttt{x.[0...i} - 1\texttt{]})|$$

Remarque. Dans la question 30, il semble que l'énoncé attende une description rapide de l'algorithme KMP (voir wikipédia pour plus de détails).

Question 31 –

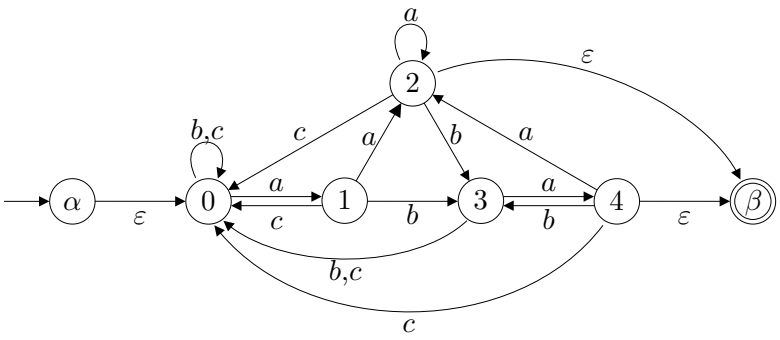
```
let automate_Q31 =
  let transition (e: int) (c: char): int = match (e,c) with
    | (0,'b') | (0,'c') | (1,'c') | (2,'c') | (3,'b') | (3,'c') | (4,'c') -> 0
    | (0,'a') -> 1
    | (1,'a') | (2,'a') | (4,'a') -> 2
    | (1,'b') | (2,'b') | (4,'b') -> 3
    | (3,'a') -> 4
    | _ -> failwith "transition: ne devrait pas arriver"
  in {
      etats = [0; 1; 2; 3];
      alphabet = ['a'; 'b'; 'c'];
      initial = 0;
      transition = transition;
      final = [2;4];
    };;
```

Question 32 – Oui, l'automate est émondé En effet :

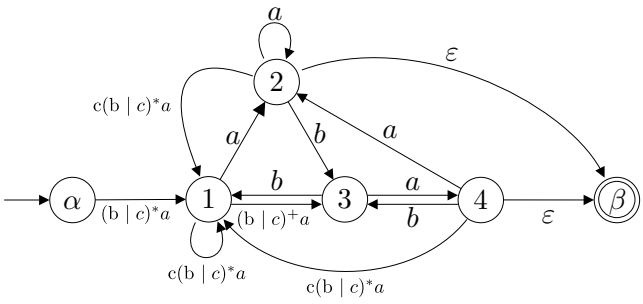
→ Tout état $e \in \{0, 1, 2, 3, 4\}$ est accessible, c'est à dire qu'il existe un chemin entre l'état initial 0 et e.

→ Tout état $e \in \{0, 1, 2, 3, 4\}$ est co-accessible, c'est à dire qu'il existe un chemin entre e et un état terminal de $T = \{2, 4\}$.

Question 33 – On commence par ajouter deux états α et β :



On élimine ensuite l'état 0 :



Question 34 – Dans l'algorithme `cherche` :

La liste `lst` contient l'ensemble des indices `i` tels que `t[0...i]` est un mot accepté par l'automate M. Avec les notations de la partie III.2 : c'est l'ensemble des indices `i` tels qu'il existe un mot de X qui soit un facteur de `t` se terminant à l'indice `i`.

Avec l'automate ci-dessus et $t = cabaac$, voici le suite des états de l'automate :

$$0, 0, 1, 3, 4, 2, 0.$$

Étant donné que les états terminaux de l'automate sont 2 et 4 :

La liste vaut `[3; 4]`

Question 35 –

```
let cherche (m: auto) (t: string): int list =
  let lst = ref [] in
  let e = ref m.initial in
  for i = 0 to String.length t - 1 do
    e := m.transition !e t.[i];
    if List.mem !e m.final then
      lst := i :: !lst
  done;
  List.rev !lst;; (* Le List.rev est facultatif *)
```

Question 36 – On utilise les notations de la partie III.2 : X est un langage non vide ne contenant pas le mot vide et M est un automate qui reconnaît le langage A^*X. Il s'agit de montrer que la fonction `cherche` appelée avec M et `t` en entrée renvoie la liste des indices `i` tels que `t[0...i]` a comme suffixe un mot de X.

Pour cela, on montre que pour tout `i` $\in [\![0, |t| - 1]\!]$, à la fin du tour de boucle `i` :

→ La variable `e` contient l'état de M après lecture du mot `t[0...i]`.

→ La variable `lst` contient la liste des indices `j` $\leqslant$ `i` tels que `t[0...j]` a comme suffixe un mot de X.

Ces propriétés se montrent facilement par itération finie sur `i`.

On en déduit qu'à la fin de la fonction (c'est à dire à la fin du tour de boucle `i = |t|-1`), la variable `lst` contient tous les indices `j` tels que `t[0...j]` a comme suffixe un mot de X.