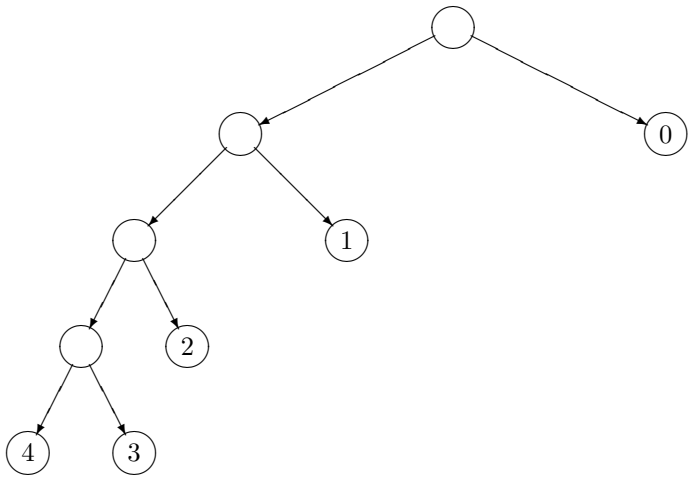
Concours blanc MP* 2022-2023

OPTION INFORMATIQUE

Corrigé

Partie I. Arbres peignes (E3A 2017)

1. Un peigne rangé à cinq feuilles est par exemple :



2. Montrons par récurrence sur $n \geq 1$ que la hauteur d'un peigne rangé à n feuilles est égale à $n-1$.
   - Un peigne rangé à 1 feuille es une feuille et sa hauteur vaut 0.
   - Supposons le résultat vrai jusqu'à un rang $n \geq 1$. Un peigne rangé à $n+1$ feuilles s'écrit `Noeud(g,d)` avec `d` qui est une feuille et `g` qui est un peigne rangé à n feuilles. La hauteur de cet arbre est égale à 1 plus le maximum des hauteurs de `g` et `d` qui valent $n-1$ (hypothèse de récurrence) et 1. Notre arbre est donc de hauteur n, ce qu'il fallait prouver.

Un peigne rangé à n feuilles est de hauteur $n-1$

3. Si l'arbre n'est pas une feuille, on vérifie qu'il y a une feuille à droite et, récursivement, que le fils gauche est un peigne rangé.

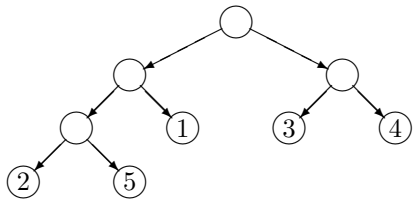
```
let rec est_range a = match a with
   |Feuille x -> true
   |Noeud (g,Feuille x) -> est_range g
   |_ -> false ;;
```

4. Si l'arbre n'est pas une feuille, on vérifie que l'un des fils est une feuille et, récursivement, que l'autre est un peigne strict.

```
let rec est_peigne_strict a = match a with
   |Feuille x -> true
   |Noeud(g,Feuille x) -> est_peigne_strict g
   |Noeud(Feuille x,d) -> est_peigne_strict d
   |_ -> false;;
```

```
let est_peigne a = match a with
   |Feuille x -> true
   |Noeud (g,d) -> (est_peigne_strict g)&&(est_peigne_strict d);;
```

5. (a) Une rotation sur l'arbre exemple donne



(b) La rotation est possible dès que le fils droit est un noeud qui a au moins un fils qui est une feuille.

```
let rotation a = match a with
   |Noeud(a1,Noeud(a2,Feuille f))->Noeud(Noeud(a1,Feuille f),a2)
   |Noeud(a1,Noeud(Feuille f,a2))->Noeud(Noeud(a1,Feuille f),a2)
   |_ -> a;;
```

(c)
```
let rec range a = match a with
   |Feuille f -> Feuille f
   |Noeud(g,Feuille f) -> Noeud(range g,Feuille f)
   |_ -> range (rotation a) ;;

let rangement a =
    if est_peigne a then range a
    else a;;
```

Partie II. Logique et calcul des propositions (CCP 2016)

1. On considère une tri-valuation f telle que $f(A) = ?$. On a alors $\widehat{f}((A \vee \neg A) = ?$ d'après les tables de vérité. $A \vee \neg A$ n'est donc pas une tautologie.
2. $A \Rightarrow A$ est une tautologie. En effet, pour une tri-valuation f, que $f(A)$ ait pour valeur $\top, \bot$ ou ?, on a toujours $\widehat{f}(A \Rightarrow A) = \top$.
3. $\widehat{f}(A \wedge B) = max(\widehat{f}(A), \widehat{f}(B))$;
   $\widehat{f}(A \vee B) = min(\widehat{f}(A), \widehat{f}(B))$.
4. On suppose que, par analogie avec la logique propositionnelle classique, deux formules sont équivalentes si chaque tri-valuation leur donne la même valeur.

   Soit f tri-valuation vérifiant $f(A) = f(B) = ?$. On a :
   - $\widehat{f}(\neg A \vee B) = ?$
   - $\widehat{f}(A \Rightarrow B) = \top$

   Ces deux formules ne sont donc pas équivalentes.
5. On vérifie aisément par la table de vérité que ces formules sont équivalentes.
6. En dressant la table, on observe que cette formule n'est pas une tautologie :
   Si $f(A) = ?$ et $f(B) = \bot$, $\widehat{f}(((A \Rightarrow B) \wedge (\neg A \Rightarrow B)) \Rightarrow B) = ?$.
7. On considère une tri-valuation f telle que $f(A) = ?$. On a alors $\widehat{f}(A \to A) = ?$ d'après les tables de vérité. Cette formule n'est donc pas une tautologie.
8. Soit f la tri-valuation affectant ? à chaque variable. On montre que $\widehat{f}(\phi) = ?$ pour toute formule ϕ par récurrence structurelle sur ϕ :

— Si ϕ est réduite à une variable propositionnel, le résultat est garanti par hypothèse sur f.
— Si $\phi = \psi \wedge \theta$, $\widehat{f}(\psi) = \widehat{f}(\theta) =$? par hypothèse de récurrence, donc $\widehat{f}(\phi) =$? d'après la table de vérité.
— Similairement, si $\phi = \psi \vee \theta$ ou $\phi = \psi \to \theta$ ou $\phi = \neg\psi$, on a $\widehat{f}(\phi) =$?.

9. **Remarque :** La proposition en question n'est **pas** une formule de la logique étudiée, demander si elle est une tautologie n'a donc pas de sens. On va se contenter de montrer qu'elle est fausse. Par ailleurs la nature de A et B n'est pas claire ; on va supposer que ce sont des formules quelconques.

   D'après la question précédente, $A \to B$ ne peut pas être une tautologie. En revanche, B peut être une conséquence de $\{A\}$ (par exemple si $B = A$). Les deux affirmations ne sont donc pas équivalentes (au sens classique du mot).

10. `Implique(Et(Implique(Var("A"),Var("B")),Implique(Non(Var("A")),Var("B"))),Var("B"))`

11. **Remarque :** L'énoncé ne mentionne rien sur les parenthèses. On peut légitimement penser qu'elles sont nécessaires dans le résultat pour éviter les ambiguïtés de lecture, mais l'exemple donné n'en contient pas. On choisit donc de ne pas en mettre.

```
let rec lectureformule phi =
        match phi with
                Var(s) -> s
                |Vrai -> "Vrai"
                |Faux -> "Faux"
                |Indéterminé ->"Indéterminé"
                |Non(psi) -> "Non "^(lectureformule psi)
                |Et(psi,theta) -> (lectureformule psi)^" et "^(lectureformule theta)
                |Ou(psi,theta) -> (lectureformule psi)^" ou "^(lectureformule theta)
                |Implique(psi,theta) -> (lectureformule psi)^" Implique "^(lectureformule theta)
```

Partie III. Langages et automates (Mines 2017)

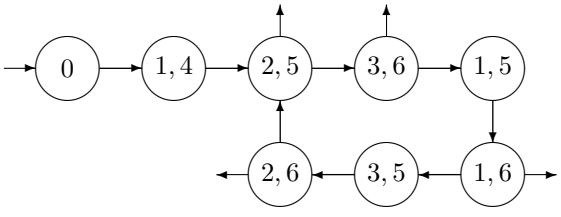
1. $L(\alpha, \beta)$ est fini si et seulement si $\alpha = 0$. De plus, $L(0, \beta) = \{a^\beta\}$ est de cardinal 1.
2. Le langage reconnu par l'automate proposé est $L(4, 2)$.
3. On a cette fois $L_2 = L(3, 2) \cup L(2, 3)$.
4. La table de transition de A_2 est la suivante :

	0	1	2	3	4	5	6
a	1, 4	2	3	1	5	6	5

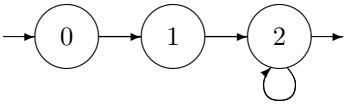
   La table du déterminisé est alors (en ne faisant apparaître que les états accessibles)

	0	1, 4	2, 5	3, 6	1, 5	2, 6	3, 5	1, 6
a	1, 4	2, 5	3, 6	1, 5	2, 6	3, 5	1, 6	2, 5

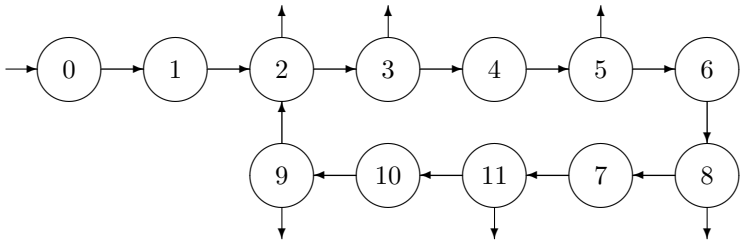
   0 est l'état initial et les états terminaux sont ceux contenant 2 ou 6, c'est à dire $(2, 5)$, $(3, 6)$, $(2, 6)$ et $(1, 6)$. On vérifie que tous les états sont co-accessibles, c'est à dire que l'automate est émondé. L'automate est le suivant :



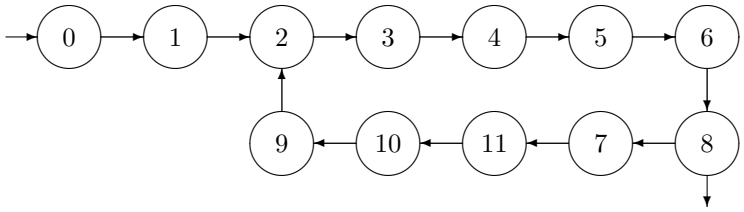
5. Le langage reconnu par A_3 est la réunion des langages des mots menant de l'état initial à chacun des états terminaux :
$$L_3 = L(6, 2) \cup L(6, 3) \cup L(6, 5) \cup L(6, 7)$$
6. L'automate suivant reconnaît $L(1, 2)$ et est de la forme F :



7. Sur le même principe que la question 4 :



8. On change les états terminaux



Le langage reconnu par cet automate montre que

$$L(2,3) \cap L(5,2) = L(10,7)$$

Remarque : on pourrait retrouver mathématiquement ce résultat en cherchant les entiers a et b tels que $2a+3 = 5b+2$, ce qui amène à l'équation diophantienne $5b - 2a = 1$.

9. Soit A un automate déterministe émondé dont la fonction de transition est notée δ (c'est une application définie d'une partie de $Q \times \{a\}$ dans Q où Q est l'ensemble des états).
Notons q_0 l'état initial de A. Distinguons deux cas.
   - Si on peut définir la suite $(q_i)_{i\in\mathbb{N}}$ par $q_{i+1} = \delta(q_i, a)$ (pour chaque état atteint, il existe un successeur), ce qui revient à dire (puisque l'automate est émondé et que tous les états sont donc accessibles) que l'automate est complet.
   Comme Q est fini, il existe deux q_i égaux. Notons s le premier s tel que $q_{s+1} \in \{q_0, \dots, q_s\}$ (s existe avec l'hypothèse faite).
   Il existe $r \in \{0, \dots, s\}$ tel que $q_{s+1} = q_r$. L'automate est alors de la forme F avec les mêmes notations que celles qui suivent la question 5.
   - Sinon, on note s le premier entier tel que q_s n'a pas de successeur. L'automate est alors le suivant (sans les états terminaux)

   Notons que puisque l'automate est émondé, q_s doit être terminal. Dans ce cas, le langage est fini et son cardinal égal au nombre des états terminaux.

Si le langage reconnu est infini, on est forcément dans le premier cas et l'automate est de la forme F.

On se donne maintenant un automate de la forme F. Adoptons les notations de l'énoncé dans le dessin qui suit la question 5. Si aucun des états $q_r, \dots, q_s$ n'est terminal alors un mot reconnu est de longueur $\leq r-1$ et le langage reconnu est fini. Sinon, en notant q_i avec $r \leq i \leq s$ un état terminal, tout mot du type $a^{i+k(s-r+1)}$ est reconnu et le langage est infini. Une CNS pour que le langage soit infini est donc qu'il existe un état terminal au moins parmi $q_r, \dots, q_s$.

10. Si L est rationnel unaire infini, il est reconnu par un automate déterministe émondé, et on vient de voir que cet automate est alors de la forme F et, avec les notations précédentes, L contient donc $L(s-r+1, i)$.
11. Supposons, par l'absurde, que L soit rationnel. L est infini car $\forall n \geq 1,\ u_{n+1} - u_n > u_1 - u_0 \geq 0$ et donc $u_{n+1} > u_n$ (les a^{u_n} pour $n \geq 1$ sont donc deux à deux distincts). La question précédente donne l'existence de $\alpha \geq 1$ et $\beta > 0$ tels que $L(\alpha, \beta) \subset L$.
Les mots de L (sauf a^{u_0}) classés par longueur strictement croissante sont $a^{u_1}, a^{u_2}, \dots$. Comme $(u_{n+1} - u_n)$ est strictement croissante, on a par récurrence

$$\forall n \in \mathbb{N}^*,\ u_{n+1} - u_n \geq n$$

A partir d'un certain rang n, on aura $u_{n+1} - u_n \geq \alpha + 1$. Il n'existe donc qu'un nombre fini de couples (m, m') de mots de l tels que $|m| - |m'| = \alpha$. ceci contredit à l'évidence $L(\alpha, \beta) \subset L$.

12. $(n+1)^2 - n^2 = 2n+1$ est le terme général d'une suite strictement croissante positive. De plus, (n^2) est une suite d'entiers positifs ou nuls. La question précédente indique que $\{a^{n^2} / \ n \in \mathbb{N}\}$ n'est pas rationnel.

Partie IV. Graphe du web (Mines 2016)

Fonctions utilitaires

1.
```
let rec aplatir liste =
     match liste with
    |[] -> []
    |(x,l)::q -> (x::l)@(aplatir q);;
```

2.
```
let rec decouper l =
     match l with
    |[] -> [],[]
    |[x] -> [x],[]
    |x::y::q -> let (l1,l2)=decouper q in
                x::l1,y::l2 ;;

let rec fusion l1 l2 =
    match (l1,l2) with
   |[],_ -> l2
   |_,[] -> l1
   |(a,b)::q1,(c,d)::q2 -> if b>=d then (a,b)::(fusion q1 l2)
                           else (c,d)::(fusion l1 q2);;

let rec tri_fusion l =
    match l with
   |[] -> []
   |[x] -> [x]
   |_  -> let (l1,l2)=decouper l in
          fusion (tri_fusion l1) (tri_fusion l2);;
```

3. On écrit une fonction auxiliaire
`uni_aux : string liste` $\to$ `dictionnaire` $\to$ `int` $\to$ `string list * dictionnaire`.
Dans l'appel `uni_aux liste dico n`, on suppose que le dictionnaire *dico* possède exactement n éléments. On renvoie un couple formé des éléments distincts de *liste* et du dictionnaire obtenu en ajoutant ces éléments comme clefs à *dico*. L'argument n sert pour savoir quelle valeur associer à une clef et respecter la consigne de l'énoncé.

```
let unique liste =
    let rec uni_aux liste dico n=
       match liste with
      |[] -> [],dico
      |c::q -> if contient c dico then uni_aux q dico n
               else let (l,d)=uni_aux q (ajoute c n dico) (n+1)
                    in (c::l,d)
    in uni_aux liste (dictionnaire_vide ()) 0;;
```

4. Pour chaque élément de *liste*, on teste son appartenance à *dico*. Le dictionnaire contenant au plus m clefs, ces tests ont un coût $O(n\log(m))$.
La construction du dictionnaire (appels à `ajoute`) a un coût $O(\sum_{i=1}^{m}\log(i)) = O(m\log(m))$.
La construction de la liste se fait par consages et a un coût $O(m)$.
Le coût total de l'appel est donc (puisque $n \geq m$) $O(n\log(m)+m)$.

Crawler simple

5. Dans une stratégie de parcours en largeur d'un graphe, on utilise usuellement :
   - un marquage des sommets pour éviter les trajets infinis (on ne redémarre pas une recherche à partir d'un sommet marqué, c'est à dire un sommet déjà visité) ;
   - une structure de file (FIFO) pour privilégier le traitement des sommets à courte distance.

Ici, le sujet nous propose à l'évidence d'utiliser un dictionnaire comme structure de marquage : un adresse est marquée si elle est enregistrée dans le dictionnaire. Notons que la valeur (quantité entière) associée aux clefs n'a a priori pas d'importance. Utiliser le type `dictionnaire` est un peu délicat car il n'est pas mutable. Il n'est donc pas question de "faire évoluer" un dictionnaire (ou il faudrait une référence de dictionnaire). Ceci explique (voir plus bas) la présence d'un élément de type dictionnaire dans les arguments de la fonction écrite.
Aucune structure n'est proposée pour modéliser les files. On pourrait utiliser la structure de file implémentée en Caml dans le module `queue`. Celui-ci n'est cependant a priori pas au programme. On pourrait aussi commencer par créer un type `file` dédié (par exemple, implémenter de manière persistante une file à l'aide d'un couple de listes). Ceci ne semble pas raisonnable en temps limité. Je choisis donc d'implémenter une file de manière persistante à l'aide d'une simple liste. La tête de la liste est la tête de la file et l'ajout d'élément doit se faire par la droite, c'est à dire par concaténation. Là encore, on aura des files persistantes (non mutables) et ceci explique la présence de la file dans les arguments (on aurait aussi pu utiliser une référence de liste).
Le dernier souci est que l'on ne doit pas faire un parcours en largeur complet. On doit l'interrompre quand on a visité n sites, c'est à dire quand le dictionnaire contient n éléments. On pourrait gérer une référence d'entier donnant le nombre de sites visités. J'ai préféré ajouter un argument entier à ma fonction donnant le nombre de site qu'il reste à ajouter (n au départ).
J'écris finalement une fonction
`remplir : int → string list → (string * string list) list → dictionnaire`
`→ (string * string list) list`
Dans l'appel `remplir k file res dico`, les différents arguments ont été expliqués plus haut. `res` correspond à la liste en construction (que l'on renvoie quand on a terminé). On pourrait éviter cet argument supplémentaire (voir la fonction de la question suivante pour varier les plaisirs).

```
let crawler_bfs n u =
    let rec remplir k file res dico =
        match file with
       |[] -> res
       |c::q -> if k=0 then res
                else if not(contient c dico) then begin
                          let l=recupere_liens c in
                          remplir (k-1) (q@l) ((c,l)::res) (ajoute c 0 dico)
                          end
                else remplir k q res dico
    in remplir n [u] [] (dictionnaire_vide());;
```

6. Le plus simple et le plus efficace pour un parcours en profondeur est de procéder récursivement, en utilisant des références :

```
let crawler_dfs n u =
        let k = ref 0 in
        let dico = ref dictionnaire_vide() in
        let res = ref [] in
        let rec visiter s =
                let l = recupere_liens s in
                dico : = ajoute s 0 !dico;
                res := (s,l)::res;
                incr k;
                let rec aux l =
                        match l with
                [] -> ()
                |x::q -> (if !k < n && not (contient !dico x)
                          then visiter x );
                          aux q
                in
                aux l
    in
    visiter u;
    List.rev !res ;;
```

7. Les fonctions `aplatir` et `unique` permettent d'obtenir la liste des sommets du graphe (naturellement numérotés par leurs positions dans la liste) et un dictionnaire permettant s'associer une URL au numéro

du sommet du graphe (connaissant une URL, on n'a pas à parcourir la liste pour connaître le numéro du sommet associé : il est donné par le dictionnaire).
Pour chaque couple (c, l) du crawl, il faut ajouter au graphe l'arc de c vers chaque élément de l (du numéro de c vers le numéro de chaque élément de l). On doit donc disposer d'une fonction
`miseajour : int → (string list) → unit`
qui prend en argument un entier (le numéro de c) et une liste (l) et qui ajoute au graphe ces arcs.
La fonction `parcourt : (string * string list) list → unit` est la fonction récursive de parcours du crawl.

```
let construit_graphe crawl =
    let (s,dico)=unique (aplatir crawl) in
    let n=List.length s in
    let g=Array.make_matrix n n 0 in
    let rec miseajour i l =
        match l with
        [] -> ()
       |ch::q -> let j=valeur ch dico in
                 g.(i).(j) <- g.(i).(j) + 1 ;
                 miseajour i q
    in
    let rec parcourt crawl =
       match crawl with
       [] -> ()
      |(c,l)::q -> let i=valeur c dico in
                   miseajour i l ;
                   parcourt q
    in parcourt crawl;
    s,g;;
```

On pourrait utiliser la fonction `List.iter` pour effectuer l'itération sur les éléments des listes.

Calcul de PageRank

Dans toute cette partie, on travaille avec des éléments de type `float`. On devra donc utiliser les opérateur arithmétiques sur les flottants et transformer tous les entiers en flottants. Cela alourdit considérablement le code.

8. Il suffit de suivre les consignes.

```
let surf_aleatoire d G =
    let n=Array.length G in
    let M=Array.make_matrix n n 0. in
    for i=0 to n-1 do
       let k=ref 0 in
       for j=0 to n-1 do k:= !k+G.(i).(j) done ;
       if !k=0 then
            for j=0 to n-1 do M.(i).(j) <- 1./.(float_of_int n) done
       else
            for j=0 to n-1 do
                  M.(i).(j) <- (1.-.d)*.(float_of_int G.(i).(j))
                                  /.(float_of_int !k)+.d/.(float_of_int n)
            done ;
       done ;
    M;;
```

9.
```
let multiplie v M =
    let n=Array.length v in
    let w=Array.make n 0. in
    for j=0 to n-1 do
       for i=0 to n-1 do
           w.(j) <- w.(j)+.v.(i)*.M.(i).(j)
           done;
       done;
    w;;
```

10. On commence par écrire une fonction de calcul de la norme de la différence de deux vecteurs (distance entre ces vecteurs).

```
let norme v w =
   let k=ref 0. in
   for i=0 to (Array.length v - 1) do
       if v.(i)-.w.(i) > 0. then k:= !k +. v.(i) -. w.(i)
       else k:= !k +. w.(i) -. v.(i)
       done ;
   !k;;
```

On écrit une fonction auxiliaire `itere` qui prend en argument un vecteur `v` et réalise le processus à partir de ce vecteur. Il suffit de l'appeler avec le bon vecteur initial.

```
let pagerank theta M =
    let rec itere v =
       let w=multiplie v M in
       if (norme v w) <= theta then w
       else itere w
    in let n=Array.length M
    in itere (Array.make n (1./.(float_of_int n))) ;;
```

11. La fonction auxiliaire `construireliste` prend en argument la liste des sommets (`string list`) et renvoie la liste (non ordonnée) des couples sommet/score, le score étant contenu dans le vecteur v obtenu avec `pagerank`. Il faut pouvoir, quand on rencontre un sommet sous forme d'URL, connaître le numéro du sommet. C'est pourquoi la fonction auxiliaire a un second argument de type `int` donnant le numéro du prochain sommet à traiter.

```
let calcule_pagerank d theta crawl =
    let (s,G)=construit_graphe crawl in
    let M=surf_aleatoire d G in
    let v=pagerank theta M in
    let rec construireliste s i =
        match s with
       |[] -> []
       |c::q -> (c,v.(i))::(construireliste q (i+1))
    in tri_fusion (construireliste s 0);;
```