Concours blanc MP* 2022-2023

OPTION INFORMATIQUE

DUREE 4h

Partie I. Arbres peignes (E3A 2017)

On suppose défini le type arbre de la manière suivante :

```
type arbre =
   |Feuille of int
   |Noeud of arbre * arbre ;;
```

On dit qu'un arbre est un *peigne* si tous les noeuds à l'exception éventuelle de la racine ont au moins une feuille pour fils. On dit qu'un peigne est un *strict* si sa racine a au moins une feuille pour fils, ou s'il est réduit à une feuille. On dit qu'un peigne est *rangé* si le fils droit d'un noeud est toujours une feuille. Un arbre réduit à une feuille est un peigne rangé.

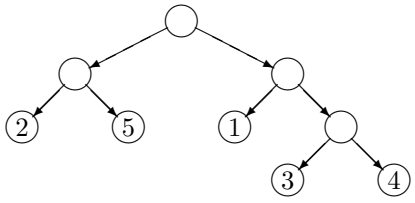


Figure 1 : un peigne à cinq feuilles

1. Représenter un peigne rangé à 5 feuilles.
2. La *hauteur* d'un arbre est le nombre de noeuds maximal que l'on rencontre pour aller de la racine à une feuille (la hauteur d'une feuille seule est 0). Quelle est la hauteur d'un peigne rangé à n feuilles ? On justifiera la réponse.
3. Ecrire une fonction `est_range : arbre` $\to$ `bool` qui renvoie `true` si l'arbre donné en argument est un peigne rangé.
4. Ecrire une fonction `est_peigne_strict : arbre` $\to$ `bool` qui renvoie `true` si l'arbre donné en argument est un peigne strict. En déduire une fonction `est_peigne : arbre` $\to$ `bool` qui renvoie `true` si l'arbre donné en argument est un peigne.
5. On souhaite ranger un peigne donné. Supposons que le fils droit N de sa racine ne soit pas une feuille. Notons A_1 le sous-arbre gauche de la racine, f l'une des feuilles du noeud N et A_2 l'autre sous-arbre du noeud N. On va utiliser l'opération de *rotation* qui construit un nouveau peigne où
   - le fils droit de la racine est le sous-arbre A_2 ;
   - le fils gauche de la racine est un noeud de sous-arbre gauche A_1 et de sous-rabre droit la feuille f.

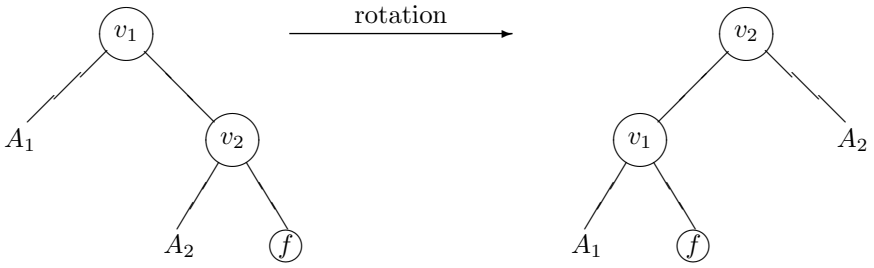


Figure 2 : une rotation

(a) Donner le résultat d'une rotation sur l'arbre de la figure 1.

(b) Ecrire une fonction `rotation : arbre` $\rightarrow$ `arbre` qui effectue l'opération décrite ci-dessus. La fonction renverra l'arbre initial si une rotation n'est pas possible.

(c) Ecrire une fonction `rangement : arbre` $\rightarrow$ `arbre` qui range un peigne donné en argument, c'est à dire qu'il renvoie un peigne rangé ayant les mêmes feuilles que celui donné en argument. La fonction renverra l'arbre initial si celui-ci n'est pas un peigne.

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

Nous nous intéressons dans cet exercice à l'étude de quelques propriétés de la logique propositionnelle tri-valuée. En plus des deux valeurs classiques VRAI ($\top$) et FAUX ($\bot$) que peut prendre une expression, la logique propositionnelle tri-valuée introduit une troisième valeur INDETERMINE (?) .
$\mathcal{V}$ est l'ensemble des variables propositionnelles et $\mathcal{F}$ l'ensemble des formules construites sur $\mathcal{V}$. Pour $A, B \in \mathcal{V}$, les tables de vérités des opérateurs classiques dans cette logique propositionnelle sont les suivantes :

(a) $A \wedge B$

A	B	$A \wedge B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	?	?
$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$
$\bot$	?	$\bot$
?	$\top$	?
?	$\bot$	$\bot$
?	?	?

(b) $A \vee B$

A	B	$A \vee B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\top$	?	$\top$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\bot$
$\bot$	?	?
?	$\top$	$\top$
?	$\bot$	?
?	?	?

(c) $A \Rightarrow B$

A	B	$A \Rightarrow B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	?	?
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$
$\bot$	?	$\top$
?	$\top$	$\top$
?	$\bot$	?
?	?	$\top$

(d) $\neg A$

A	$\neg A$
$\top$	$\bot$
$\bot$	$\top$
?	?

Définitions

Définition 1 Tri-valuation.
Une tri-valuation est une fonction $f : \mathcal{V} \rightarrow \{\top, \bot, ?\}$.

On étend alors de manière usuelle la notion de tri-valuation sur l'ensemble des formules :

Définition 2 Une tri-valuation sur l'ensemble des formules est une fonction $\widehat{f} : \mathcal{F} \rightarrow \{\top, \bot, ?\}$.

Définition 3 Une tri-valuation $\widehat{f}$ satisfait une formule ϕ si $\widehat{f}(\phi) = \top$. On notera alors $\widehat{f} \vdash_3 \phi$.

Définition 4 Formule.
Une formule ϕ est :

— une conséquence d'un ensemble de formules $\mathcal{X}$ si toute interprétation qui satisfait toutes les formules de $\mathcal{X}$ satisfait ϕ. On notera dans ce cas $\mathcal{X} \Vdash_3 \phi$;

— une tautologie si pour toute tri-valuation $\widehat{f}$, $\widehat{f}(\phi) = \top$. On notera dans ce cas $\Vdash_3 \phi$.

Questions

1. Montrer que $A \vee \neg A$ n'est pas une tautologie.

2. Proposer alors une tautologie simple dans cette logique.
   Posons $\top = 1$, $\bot = 0$ et $? = 0,5$.
3. Proposer un calcul simple permettant de trouver la table de vérité de $A \wedge B$ en fonction de A et B. Même question pour $A \vee B$.
4. En logique bi-valuée classique, les propositions $\neg A \vee B$ et $A \Rightarrow B$ sont équivalentes. Qu'en est-il dans le cadre de la logique propositionnelle tri-valuée ?
5. En écrivant les tables de vérité, indiquer si les propositions $\neg B \Rightarrow \neg A$ et $A \Rightarrow B$ sont équivalentes.
6. Donner la table de vérité de la proposition $((A \Rightarrow B) \wedge (\neg A \Rightarrow B)) \Rightarrow B$. Cette proposition est-elle une tautologie ?
   Un nouvel opérateur d'implication, noté $\rightarrow$, est alors défini, dont la table de vérité est la suivante :

A	B	$A \rightarrow B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	$?$	$?$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\top$
$\bot$	$?$	$\top$
$?$	$\top$	$\top$
$?$	$\bot$	$?$
$?$	$?$	$?$

7. $A \rightarrow A$ est-il une tautologie ?
8. Montrer qu'il n'existe aucune tautologie en utilisant uniquement cette définition de l'implication.
9. La proposition suivante est-elle une tautologie :

   “ $(\{A\} \Vdash_3 B)$ est équivalent à $(\Vdash_3 A \rightarrow B)$ ” ?

   On définit alors un type **FormuleLogique** représentant les formules de la manière suivante :

```
type Formulelogique
| Vrai (* Constante Vrai *)
| Faux (* Constante Faux *)
| Indetermine (* Constante Indeterminé *)
| Var of string (* Variable propositionnelle *)
| Non of FormuleLogique (* Négation d'une formule *)
| Et of FormuleLogique*FormuleLogique (* conjonction de deux formules *)
| Ou of FormuleLogique*FormuleLogique (* disjonction de deux formules *)
| Implique of FormuleLogique*FormuleLogique (* implication *)
```

10. Avec la représentation précédente, écrire en CaML la formule :

$$((A \Rightarrow B) \wedge (\neg A \Rightarrow B)) \Rightarrow B.$$

11. Écrire alors une fonction récursive CaML **lectureFormule**, prenant en argument une formule et renvoyant une chaine de caractères spécifiant comment un lecteur lirait la formule. Ainsi, par exemple, pour $\phi = A \wedge (\neg B)$, **lectureFormule** renvoie **A et non B**.

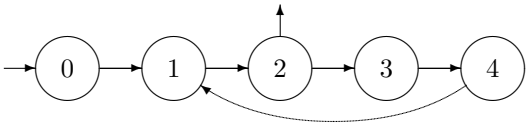
Partie III. Langages et automates (Mines 2017)

On s'intéresse aux langages sur l'alphabet $\Sigma = \{a\}$; un tel langage est dit *unaire*. Un automate reconnaissant un langage unaire sera dit *unaire*. Lorsqu'on dessinera un automate unaire, il ne sera pas utile de faire figurer les étiquettes des transitions, toutes ces étiquettes étant l'étiquette a. C'est ce qui est fait dans cet énoncé.
Dans un automate unaire, on appelle *chemin* une suite $q_1, \ldots, q_p$ d'états telle que, pour i compris entre 1 et p, il existe une transition de q_{i-1} vers q_i ; on dit qu'il s'agit d'un chemin de q_1 à q_p. On appelle *circuit* un chemin $q_1, \ldots, q_p$ tel qu'il existe une transition de q_p vers q_1.
Dans cet exercice, tous les automates considérés seront finis et auront un et un seul état initial. On dit qu'un automate est *émondé* si, pour tout état q, il existe d'une part un chemin de l'état initial à q et d'autre part un chemin de q à un état final.
On rappelle qu'un langage non vide est rationnel si et seulement s'il est reconnu par un automate ou encore si et seulement s'il est reconnu par un automate déterministe émondé.

Soient α et β deux entiers positifs ou nuls. On note $L(\alpha, \beta)$ le langage unaire défini par :

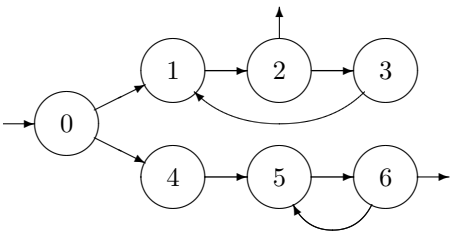
$$L(\alpha, \beta) = \{a^{\alpha k+\beta} / \ k \in \mathbb{N}\}$$

1. Donner sans justification une condition nécessaire et suffisante pour que $L(\alpha, \beta)$ soit fini. Dans le cas où cette condition est satisfaite, donner sans justification le cardinal de $L(\alpha, \beta)$.
2. On considère l'automate A_1 ci-dessous. Indiquer sans justification deux entiers α_1, β_1 tels que A_1 reconnaisse le langage $L(\alpha_1, \beta_1)$.



Automate A_1

3. On considère l'automate A_2 ci-dessous :

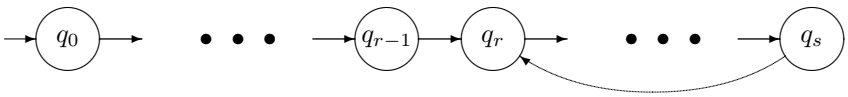


Automate A_2

On note L_2 le langage reconnu par A_2. Indiquer sans justification quatre entiers $\alpha_2, \beta_2, \alpha_3, \beta_3$ tels que A_2 reconnaisse le langage $L_2 = L(\alpha_2, \beta_2) \cup L(\alpha_3, \beta_3)$.

4. Construire un automate *déterministe émondé* A_3 en appliquant la procédure de déterminisation à l'automate A_2.
5. En s'appuyant sur l'automate A_3, indiquer sans justification cinq entiers $\alpha_4, \beta_4, \beta_5, \beta_6, \beta_7$ tel que A_3 reconnaisse le langage $L_3 = L(\alpha_4, \beta_4) \cup L(\alpha_4, \beta_5) \cup L(\alpha_4, \beta_6) \cup L(\alpha_4, \beta_7)$ (remarque : le langage L_3 est égal par ailleurs au langage L_2).

On dit ci-dessous qu'un automate est *de la forme* F si, en omettant les états finals, il peut se tracer selon le schéma ci-dessous :



Le chemin $q_0, \ldots, q_{r-1}$ peut être vide, auquel cas on a $r = 0$. Le circuit $q_r, \ldots, q_s$ ne doit pas être vide mais on peut avoir $r = s$ avec une transition de l'état q_r vers lui même (un tel circuit s'appelle aussi une *boucle*). On constate que les automates A_1 et A_3 sont de la forme F, mais non A_2.

6. Dessiner sans justification un automate de la forme F qui reconnaît le langage $L(1, 2)$. On fera figurer le ou les états finals.
   ATTENTION : on ne demande aucune justification mais uniquement de tracer un automate de la forme F en choisissant correctement les longueurs du chemin et du circuit et en ajoutant le ou les état(s) final(s).
7. Dessiner un automate de la forme F qui reconnaît le langage $L(2, 3) \cup L(5, 2)$. On fera figurer le ou les état(s) final(s). Comme à la question précédente, on ne demande aucune justification.
8. En s'inspirant de la réponse à la question précédente, décrire sans justification un automate de la forme F qui reconnaît le langage $L(2, 3) \cap L(5, 2)$. Indiquer deux entiers α et β tels que $L(2, 3) \cap L(5, 2) = L(\alpha, \beta)$.
9. Montrer qu'un automate déterministe émondé qui reconnaît un langage unaire rationnel infini est de la forme F. Donner une condition nécessaire et suffisante portant sur les états finals pour qu'un automate de la forme F reconnaisse un langage infini.
10. Soit L un langage rationnel unaire infini. En s'appuyant sur la question précédente, montrer qu'il existe deux entiers $\alpha \geq 1$ et $\beta \geq 0$ tels que L contient $L(\alpha, \beta)$.
11. On considère une suite $(u_n)_{n\geq 0}$ de nombres entiers positifs ou nuls. On suppose que la suite $(u_{n+1} - u_n)_{n\geq 0}$ est positive et strictement croissante. Soit L le langage défini par

$$L = \{a^{u_n} / \ n \geq 0\}$$

    En utilisant la question précédente, montrer que L n'est pas rationnel.
12. Montrer que le langage L défini par $L = \{a^{n^2} / \ n \geq 0\}$ n'est pas rationnel.

Partie IV. Graphe du web (Mines 2016)

Le World Wide Web, ou Web, est un ensemble de pages Web (identifiées de manière unique par leurs adresses Web, ou URL pour Uniform Resource Locators, de la forme http ://mines-ponts.fr/index.php) reliées les unes aux autres par des hyperliens. Le Web est souvent modélisé comme un graphe orienté dont les sommets sont les pages Web et les arcs les hyperliens entre pages. Le Web étant potentiellement infini, on s'intéresse à des sous-graphes du Web obtenus en naviguant sur le Web, c'est-à-dire en le parcourant page par page, en suivant les hyperliens d'une manière bien déterminée. Ce parcours du Web pour en collecter des sous-graphes est réalisé de manière automatique par des logiciels autonomes appelés Web crawlers ou crawlers en anglais, ou collecteurs en français.

Fonctions utilitaires

Nous allons tout d'abord coder certaines fonctions de manipulation de structures de données de base, qui seront utiles dans le reste de l'exercice.

1. Coder une fonction `aplatir : ('a * 'a list) list → 'a list`, telle que, si `liste` est une liste de couples $[(x_1, l_{x_1}); \ldots; (x_n, l_{x_n})]$, où chaque x_i est un élément de type `'a`, et l_{x_i} une liste d'éléments de type `'a` de la forme $[y_{i1}; \ldots; y_{ik_i}]$, `aplatir liste` est une liste d'éléments de type `'a`

$$[x_1; y_{11}; \ldots; y_{1k_1}; x_2; y_{21}; \ldots; y_{2k_2}; \ldots; x_n; y_{n1}; \ldots; y_{nk_n}]$$

2. Coder une fonction `tri_fusion : ('a * 'b) list → ('a * 'b) list` triant une liste de couples (x, y) par ordre décroissant de la valeur de la seconde composante y de chaque couple. On devra utiliser l'algorithme de tri par partition–fusion (aussi appelé « tri fusion »). Quelle est la complexité de cet algorithme ?

On va utiliser dans la suite de l'exercice un type de données `dictionnaire` qui permet de stocker des couples formés d'une chaîne de caractères (une clef) et d'un entier (une valeur). On dit que le dictionnaire associe la valeur à la clef. A chaque clef présente dans le dictionnaire est associée une seule valeur. Les fonctions suivantes sont supposées être prédéfinies :

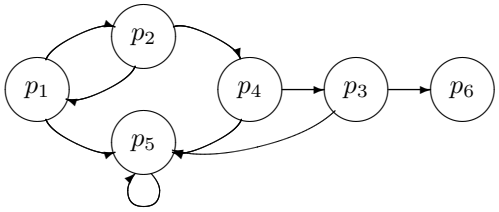
- `dictionnaire_vide : unit → dictionnaire`.
  L'appel `dictionnaire_vide ()` crée un nouveau dictionnaire vide.
- `ajoute : string → int → dictionnaire → dictionnaire`.
  L'appel `ajoute clef valeur dict` renvoie un nouveau dictionnaire identique au dictionnaire *dict*, sauf qu'un couple $(clef, valeur)$ y a été ajouté. Cette fonction s'exécute en temps $O(\log n)$ où n est le nombre d'entrées du dictionnaire.
- `contient : string → dictionnaire → bool`.
  L'appel `contient clef dict` renvoie un booléen indiquant s'il y a un couple dont la clef est $clef$ dans le dictionnaire *dict*. Cette fonction s'exécute en temps $O(\log n)$ où n est le nombre d'entrées du dictionnaire.
- `valeur : string → dictionnaire → int`.
  L'appel `valeur clef dict` renvoie la valeur associée à la clef $clef$ dans le dictionnaire *dict*. Cette fonction s'exécute en temps $O(\log n)$ où n est le nombre d'entrées du dictionnaire. Cette fonction ne peut être appelée que si la clef $clef$ est présente dans le dictionnaire.

On suppose pour la suite de l'exercice que le type de données `dictionnaire` est prédéfini ; on ne demande pas de l'implémenter.

3. Coder `unique : string list → string list * dictionnaire`, qui est telle que `unique liste` renvoie un couple $(liste', dict)$ où $liste'$ est la liste des chaînes de caractères de liste distinctes (dans l'ordre de leur première occurrence dans liste) et où *dict* associe à chaque chaîne de caractères dans $liste'$ sa position dans $liste'$ (en numérotant à partir de 0). Ainsi l'appel `unique ["x";"zz";"x";"x";"zz";"yt"]` renvoie un couple formé de la liste `["x";"zz";"yt"]` et d'un dictionnaire associant à `"x"` la valeur `0`, à `"zz"` la valeur `1` et à `"yt"` la valeur `2`.
4. Quelle est la complexité de la fonction `unique` en terme de la longueur n de la liste *liste* en argument et du nombre m d'éléments distincts dans la liste *liste* ? Justifier la réponse.

Crawler simple

Nous allons maintenant implémenter un crawler simple en Caml. On suppose fournie une fonction `recupere_liens : string → string list` prenant en argument l'URL d'une page Web p et renvoyant la liste des URL des pages q pour lesquelles il existe un hyperlien de p à q, dans l'ordre lexicographique.
Pour illustrer le comportement de cette fonction, nous considérons un exemple de mini-graphe du Web à six pages et neuf hyperliens comme suit :



Dans cette représentation, p_1, p_2, etc., sont les URL de pages Web (simplifiées pour l'exemple), et les arcs représentent les hyperliens entre pages Web.
Dans ce mini-graphe, un appel à `recupere_liens "p1"` retourne la liste `["p2"; "p5"]`.
Un crawler est un programme qui, à partir d'une URL, parcourt le graphe du Web en visitant progressivement les pages dont les liens sont présents dans chaque page rencontrée, en suivant une stratégie de parcours de graphe (par exemple, largeur d'abord, ou profondeur d'abord). A chaque nouvelle page, si celle-ci n'a pas déjà été visitée, tous ses hyperliens sont récupérés et ajoutés à une liste de liens à traiter. Le processus s'arrête quand une condition est atteinte (par exemple, un nombre fixé de pages ont été visitées). Le résultat renvoyé par le crawler, que l'on définira plus précisément plus loin, est appelé un crawl.

5. Coder `crawler_bfs : int → string → (string * string list) list` qui prend en entrée un nombre n de pages et une URL u et renvoie en sortie une liste de longueur au plus n de couples (v, l) où v est l'URL d'une page visitée (les pages apparaissant dans l'ordre où elles ont été visitées) et l la liste des liens récupérés sur la page v. On demande que `crawler_bfs` parcoure le graphe du Web en suivant une stratégie en largeur d'abord (breadth-first search), c'est-à-dire en visitant en priorité les pages rencontrées le plus tôt dans l'exploration. Le crawler doit visiter n pages distinctes, et donc appeler n fois la fonction `recupere_liens` (sauf s'il n'y a plus de pages à visiter). On utilisera une variable de type dictionnaire pour se souvenir des pages déjà visitées.
   Par exemple, sur le mini-graphe, `crawler_bfs 4 "p1"` pourra renvoyer le résultat :

```
["p1", ["p2"; "p5"];
 "p2", ["p1"; "p4"];
 "p5", ["p5"];
 "p4", ["p3"; "p5"]]
```

6. Coder `crawler_dfs : int → string → (string * string list) list` qui prend en entrée un nombre n de pages et une URL u et renvoie en sortie une liste de longueur au plus n de couples (v, l) où v est l'URL d'une page visitée (les pages apparaissant dans l'ordre où elles ont été visitées) et l la liste des liens récupérés sur la page v. On demande que `crawler_dfs` parcoure le graphe du Web en suivant une stratégie en profondeur d'abord (depth-first search), c'est-à-dire en visitant en priorité les pages rencontrées le plus récemment dans l'exploration. Le crawler doit visiter n pages distinctes, et donc appeler n fois la fonction `recupere_liens` (sauf s'il n'y a plus de pages à visiter). On utilisera une variable de type dictionnaire pour se souvenir des pages déjà visitées.
   Par exemple, sur le mini-graphe, crawler_dfs 4 "p1" pourra renvoyer le résultat :

```
["p1", ["p2"; "p5"];
 "p2", ["p1"; "p4"];
 "p4", ["p3"; "p5"];
 "p3", ["p5"; "p6"]]
```

7. Coder une fonction Caml `construit_graphe : (string * string list) list → string list * int array array` telle que si *crawl* est le résultat renvoyé par un crawler (une liste de couples formés d'une URL v et de la liste des liens récupérés sur la page v), alors `construit_graphe crawl` est un couple (l, G) où l est une liste de toutes les URL de pages contenues dans la liste *crawl* et G est la matrice d'adjacence du sous-graphe partiel du Web restreint aux pages de la liste l : G_{ij} est le nombre de liens découverts dans le crawl de la page d'indice i dans l vers la page d'indice j dans l. On fera commencer les indices à 0. Pour coder la fonction `construit_graphe`, on pourra utiliser les fonctions aplatir et unique.
   Par exemple, sur le mini-graphe, si crawl est une variable contenant le résultat de l'appel `crawler_bfs 4 "p1"` (voir question 5), alors `construit_graphe crawl` doit renvoyer :

```
["p1"; "p2"; "p5"; "p4"; "p3"],
[|[|0; 1; 1; 0; 0|];
  [|1; 0; 0; 1; 0|];
  [|0; 0; 1; 0; 0|];
  [|0; 0; 1; 0; 1|];
  [|0; 0; 0; 0; 0|]|]
```

En particulier :

- p_3 apparaît même s'il n'a pas été visité dans le crawl ;
- p_6 n'apparaît pas car il n'a pas été découvert dans le crawl ;
- l'hyperlien de p_3 à p_5 n'apparaît pas car p_3 n'a pas été visité.

Calcul de PageRank

PageRank est une manière d'affecter un score à l'ensemble des pages du Web, imaginée par Sergey Brin et Larry Page, les fondateurs du moteur de recherche Google. L'introduction de PageRank a révolutionné la technologie des moteurs de recherche sur le Web. Nous allons maintenant implémenter le calcul de PageRank.
Etant donnée une partie du Web (où l'ensemble des pages est indexé entre 0 et $n-1$), la matrice de surf aléatoire dans cette partie du Web est la matrice M de taille $n \times n$ définie comme suit :

- S'il n'y a aucun lien depuis une page Web d'indice i, alors pour tout j, $M_{ij} := 1/n$.
- Sinon, s'il y a k_i liens depuis la page Web d'indice i, alors pour tout j, on a $M_{ij} := (1-d) \times G_{ij}/k_i + d/n$, où G_{ij} est le nombre de liens depuis la page d'indice i vers la page d'indice j et d est un nombre réel fixé appartenant à $[0, 1]$ (on prend souvent $d = 0,15$).

Cette matrice peut être vue comme décrivant la marche aléatoire d'un surfeur sur le Web. à chaque fois que celui-ci visite une page Web :

- Si cette page ne comporte aucun lien, il visite une page Web arbitraire, choisie aléatoirement de façon uniforme.
- Si cette page comporte au moins un lien, il visite avec une probabilité égale à $1/d$ un des liens sortants de cette page, et avec une probabilité égale à d une page Web arbitraire, choisie aléatoirement de façon uniforme.

8. Coder `surf_aleatoire : float → int array array → float array array` telle que si d est un nombre entre 0 et 1, et si G est la matrice d'adjacence d'un sous-graphe partiel du Web, alors `surf_aleatoire d G` renvoie la matrice M de surf aléatoire dans ce sous-graphe.
9. Coder `multiplie : float array → float array array → float array`, une fonction prenant en argument un vecteur ligne v de taille n et une matrice M de taille $n \times n$ et renvoyant le vecteur ligne w de taille n résultant du produit de v par la matrice M : $w = vM$. En d'autres termes, pour tout j, $w_j = \sum_i v_i M_{ij}$.

Le PageRank des pages d'un sous-graphe du Web à n pages se calcule par des multiplications successives d'un vecteur ligne par la matrice de surf aléatoire M de ce sous-graphe. Plus précisément, soit θ un nombre réel strictement positif (par exemple, $\theta = 10^{-4}$) et soit $v^{(0)}$ le vecteur ligne de taille n dont toutes les composantes valent $1/n$. On pose pour un entier naturel p arbitraire $v^{(p)} := v^{(0)}M^p$. L'algorithme de PageRank calcule la suite des $v^{(p)}$ pour $p = 0, 1, \ldots$ jusqu'à ce que $\|v^{(p+1)} - v^{(p)}\|_1 \leq \theta$ et renvoie alors le vecteur $v^{(p+1)}$, considéré comme le vecteur des scores de PageRank. On peut montrer (à l'aide du théorème de Perron–Frobenius) que l'algorithme termine dès lors que d est strictement positif.
PageRank est utilisé pour affecter un score d'importance aux pages du Web. Le vecteur de scores v retourné par l'algorithme de PageRank donne dans v_i le score d'importance de la page d'indice i. Les pages de plus haut score de PageRank sont considérées comme les plus importantes.

10. Coder `pagerank : float → float array array → float array`, une fonction prenant en argument un nombre $\theta > 0$ et une matrice M de surf aléatoire d'un sous-graphe du Web et renvoyant le vecteur des scores de PageRank pour θ et M. La fonction `pagerank` devra faire appel à la fonction `multiplie` précédemment codée.
11. Coder `calcule_pagerank : float → float → (string * string list) list → (string * float) list` telle que `calcule_pagerank d theta crawl` renvoie une liste de couples (u, s), un couple pour chaque URL découverte dans le crawl crawl, triée par valeur décroissante de s, où u est l'URL de cette page et s son score de PageRank. Ici, d et θ sont les deux paramètres nécessaires au calcul de la matrice de surf aléatoire et du PageRank respectivement. On pourra faire appel à la fonction `tri_fusion` et à l'ensemble des fonctions développées dans les questions précédentes.