Robots

May 21, 2019

1 Table of Contents

1 Texte d'oral de modélisation - Agrégation Option Informatique
1.1 Préparation à l'agrégation - ENS de Rennes, 2017-18
1.2 À propos de ce document
1.3 Question de programmation
1.3.1 Plus ou moins de liberté dans le choix de modélisation ?
1.4 Réponse à l'exercice requis, première approche
1.4.1 Choix des structures de données
1.4.2 Fonction transition
1.4.3 Exemple
1.5 Une autre approche avec des chaînes de Markov
1.5.1 Échantillonage pondéré
1.5.2 Simuler une étape d'une chaîne de Markov ?
1.5.3 Modéliser nos robots avec des chaînes de Markov
1.5.4 Exemple 1
1.5.5 Exemple 2
1.5.6 Conclusion de cette approche par des chaînes de Markov
1.6 Conclusion

2 Texte d'oral de modélisation - Agrégation Option Informatique

2.1 Préparation à l'agrégation - ENS de Rennes, 2017-18

- *Date* : 12 janvier 2018, démonstration d'un oral d'agrégation.
- *Auteur* : Lilian Besson
- *Texte*: Annale 2008, "Robots"

2.2 À propos de ce document

- Ceci est une *proposition* de correction, partielle et probablement non-optimale, pour la partie implémentation d'un texte d'annale de l'agrégation de mathématiques, option informatique.
- Ce document est un notebook Jupyter, et est open-source sous Licence MIT sur GitHub, comme les autres solutions de textes de modélisation que j'ai écrite cette année.
- L'implémentation sera faite en OCaml, version 4+ :

```
In [1]: print_endline Sys.ocaml_version;;
        Sys.command "ocaml -version";;
```

```
4.04.2
```

```
Out[1]: - : unit = ()
```

```
The OCaml toplevel, version 4.04.2
```

```
Out[1]: - : int = 0
```

```
In [2]: let print = Printf.printf;;
```

```
Out[2]: val print : ('a, out_channel, unit) format -> 'a = &lt;fun>
```

2.3 Question de programmation

La question de programmation pour ce texte était donnée au milieu, en page 3 :

> ń On suppose donnés les tableaux T_i. Un état du système est représenté par un vecteur U de longueur n, dont la i-ème composante contient la position du robot R_i (sous forme du numéro j du lieu L_j où il se trouve). ż
>
> ń Écrire une fonction/procédure/méthode transition qui transforme U en un état suivant du système (on admettra que les données sont telles qu'il existe un état suivant). Simuler le système de robots pendant n unités de temps. On prendra comme état initial du robot R_i le premier élément du tableau T_i . ż

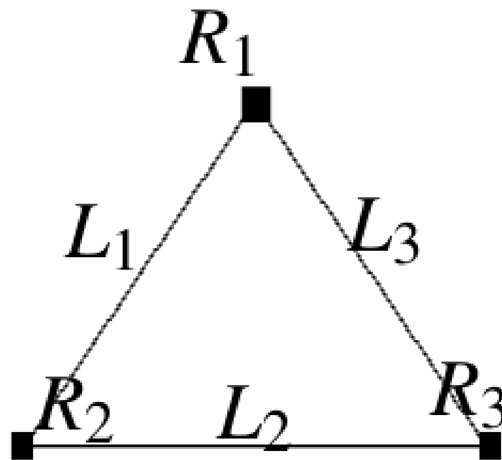
2.3.1 Plus ou moins de liberté dans le choix de modélisation ?

Vous remarquez que même avec une question bien précise comme celle-là, on dispose d'une relative liberté : la question n'impose pas le choix de modélisation !

Elle pourrait être traitée avec un automate ou graphe produit non simplifié, ou un graphe produit plus réduit (1ère solution), mais on aurait tout aussi pu utiliser une approche probabiliste, avec des chaînes de Markov par exemple (2ème solution).

2.4 Réponse à l'exercice requis, première approche

> Merci à Romain Dubourg (2017) pour son code. Cette deuxième solution est bien plus concise que l'approche avec un graphe produit non simplifié, en utilisant une approche plus directe.



Premier exemple de robots

2.4.1 Choix des structures de données

En utilisant des tableaux, `int array`, au lieu de listes, pour représenter les états u on peut modifier l'état *en place* !

```
In [4]: type etat = int array;;
        type liste_rdv = (int array) array;;

Out[4]: type etat = int array


Out[4]: type liste_rdv = int array array
```

Avec l'exemple du texte.

```
In [5]: let ex1_1 : etat = [| 1; 1; 2 |];;
        let ex1 : liste_rdv = [|
            [| 1; 3 |];
            [| 1; 2 |];
            [| 2; 3 |]
        |];;

Out[5]: val ex1_1 : etat = [|1; 1; 2|]


Out[5]: val ex1 : liste_rdv = [|[|1; 3|]; [|1; 2|]; [|2; 3|]|]
```

On peut facilement trouver la première position de `x` dans une liste et dans un tableau et `-1` sinon.

In [6]:
```
let trouve (x:int) (a:int list) : int =
    let rec aux (x:int) (a:int list) (i : int) : int =
        match a with
        | [] -> -1
        | t :: _ when (t = x) -> i
        | _ :: q -> aux x q (i+1)
    in
    aux x a 0
;;
```

Out[6]:
```
val trouve : int -> int list -> int = <fun>
```

In [7]:
```
let trouve_array (x : int) (a : int array) : int =
    trouve x (Array.to_list a)
;;
```

Out[7]:
```
val trouve_array : int -> int array -> int = <fun>
```

In [8]:
```
let _ = trouve_array 2 ex1_1;; (* 2 *)
```

Out[8]:
```
- : int = 2
```

On a besoin de pouvoir obtenir la liste des paires de robots pouvant réaliser un rendez-vous.

In [9]:
```
let rdv (u : etat) : ((int * int) list) =
    let n = Array.length u in
    let ls = ref [] in (* plus simple que d'imbriquer les List.filter et List.map... *
    for k = 0 to n - 1 do
        let i = trouve_array u.(k)
            (Array.sub u (k + 1) (n - (k + 1)))
        in
        if i >= 0 then
            ls := (k, i + k + 1) :: !ls;
    done;
    !ls
;;
```

Out[9]:
```
val rdv : etat -> (int * int) list = <fun>
```

Un rapide, pour visualiser le fonctionnement de la fonction `rdv`.

In [10]:
```
let _ = rdv ex1_1 ;;
```

Out[10]:
```
- : (int * int) list = [(0, 1)]
```

```
In [11]: let ex3_1 = [| 1; 2; 1; 4; 4 |];;
         let _ = rdv ex3_1 ;;

Out[11]: val ex3_1 : int array = [|1; 2; 1; 4; 4|]


Out[11]: - : (int * int) list = [(3, 4); (0, 2)]
```

Étant donné un état et une paire de robots pouvant réaliser un rendez-vous, la fonction suivante le réalise, en modifiant *en place* l'état u.

C'est bien plus simple que de traiter avec une approche fonctionnelle.

Pour une fonction comme ça, il faut absolument :

- utiliser des variables intermédiaires,
- et des noms de variables un peu explicites (attention aux `1`, `i`, `I` et `l` qui se ressemblent beaucoup une fois projetés au tableau !).

```
In [12]: let realise_rdv (u : etat) (lr : liste_rdv) (xy : int * int) : etat =
             let x, y = xy in
             let ux = u.(x) and uy = u.(y) in
             let rx, ry = lr.(x), lr.(y) in
             u.(x) <- rx.(((trouve_array ux rx) + 1) mod (Array.length rx));
             u.(y) <- ry.(((trouve_array uy ry) + 1) mod (Array.length ry));
             u
         ;;

Out[12]: val realise_rdv : etat -> liste_rdv -> int * int -> etat = <fun>
```

Un rapide, pour visualiser le fonctionnement de la fonction `realise_rdv`.

```
In [13]: let u = [| 0; 0; 1 |];;
         let l = [| [|0; 2|]; [|1; 2|]; [|2; 0|] |];;
         let _ = realise_rdv u l (0, 1) ;;

Out[13]: val u : int array = [|0; 0; 1|]


Out[13]: val l : int array array = [|[|0; 2|]; [|1; 2|]; [|2; 0|]|]


Out[13]: - : etat = [|2; 1; 1|]
```

On vérifie que l'état `u` a bien été modifié en place :

```
In [14]: u;;

Out[14]: - : int array = [|2; 1; 1|]
```

2.4.2 Fonction `transition`

Et enfin, on calcule l'état suivant u_1 à partir de l'état u_0, en appliquant la fonction `realise_rdv` à chaque état qui peut être modifié.

```
In [15]: let transition (u0 : etat) (l0 : liste_rdv) : etat =
             List.iter (fun u -> ignore (realise_rdv u0 l0 u)) (rdv u0);
             u0
         ;;
```

```
Out[15]: val transition : etat -> liste_rdv -> etat = <fun>
```

On effectue `n` transitions successives, non pas avec une approche récursive (qui ne serait pas récursive terminale, et donc avec une mémoire de pile d'appel linéaire en $\mathcal{O}(n)$), mais avec une simple boucle `for`.

```
In [16]: let rec n_transitions_trop_couteux (u : etat) (l : liste_rdv) (n : int) : etat =
             if (n = 0) then
                 u
             else
                 n_transitions_trop_couteux (transition u l) l (n-1)
         ;;
```

```
Out[16]: val n_transitions_trop_couteux : etat -> liste_rdv -> int -> etat = <fun>
```

```
In [17]: let n_transitions (u : etat) (l : liste_rdv) (n : int) : etat =
             for _ = 1 to n do
                 ignore (transition u l)  (* u est changé en place *)
             done;
             u
         ;;
```

```
Out[17]: val n_transitions : etat -> liste_rdv -> int -> etat = <fun>
```

2.4.3 Exemple

Avec l'exemple du texte :

```
In [18]: let _ = ex1;;
         let _ = ex1_1;;
```

```
Out[18]: - : liste_rdv = [|[|1; 3|]; [|1; 2|]; [|2; 3|]|]
```

```
Out[18]: - : etat = [|1; 1; 2|]
```

```
In [19]: let _ = transition ex1_1 ex1;;
         let _ = transition ex1_1 ex1;;
         let _ = transition ex1_1 ex1;;
         let _ = transition ex1_1 ex1;;
```

```
Out[19]: - : etat = [|3; 2; 2|]
```

```
Out[19]: - : etat = [|3; 1; 3|]
```

```
Out[19]: - : etat = [|1; 1; 2|]
```

```
Out[19]: - : etat = [|3; 2; 2|]
```

Avec l'autre exemple donné à l'oral, qui est un peu différent de celui du texte.

Quatre robots, R_0, R_1, R_2, R_3, ont comme liste de rendez-vous successifs, $T_0 = [0, 1, 2]$, $T_1 = [0]$, $T_2 = [1, 3]$ et $T_3 = [2, 3]$.

```
In [20]: let ex2 = [| [|0; 1; 2|]; [|0|]; [|1; 3|]; [|2; 3|] |];;

         let ex2_1 = [| 0; 0; 1; 2 |];;
         let _ = n_transitions ex2_1 ex2 3;;
```

```
Out[20]: val ex2 : int array array = [|[|0; 1; 2|]; [|0|]; [|1; 3|]; [|2; 3|]|]
```

```
Out[20]: val ex2_1 : int array = [|0; 0; 1; 2|]
```

```
Out[20]: - : etat = [|0; 0; 3; 3|]
```

C'est trivial, mais il peut être utile de vérifier que `n_transitions 3` fait pareil que trois appels à `transition` :

```
In [21]: let ex2_1 = [| 0; 0; 1; 2 |];; (* il faut l'écrire, il a été modifié *)
         let _ = transition ex2_1 ex2;;
         let _ = transition ex2_1 ex2;;
         let _ = transition ex2_1 ex2;;
         let _ = transition ex2_1 ex2;;
```

```
Out[21]: val ex2_1 : int array = [|0; 0; 1; 2|]
```

```
Out[21]: - : etat = [|1; 0; 1; 2|]
```

```
Out[21]: - : etat = [|2; 0; 3; 2|]

Out[21]: - : etat = [|0; 0; 3; 3|]

Out[21]: - : etat = [|1; 0; 1; 2|]
```

Avec un autre état initial :

```
In [22]: let ex2_2 = [| 0; 0; 3; 2 |];;

         let _ = transition ex2_2 ex2;;
         let _ = transition ex2_2 ex2;; (* On bloque ! *)
         let _ = transition ex2_2 ex2;; (* On bloque ! *)

Out[22]: val ex2_2 : int array = [|0; 0; 3; 2|]

Out[22]: - : etat = [|1; 0; 3; 2|]

Out[22]: - : etat = [|1; 0; 3; 2|]

Out[22]: - : etat = [|1; 0; 3; 2|]
```

Et avec encore un autre état initial :

```
In [23]: let ex2_3 = [| 0; 0; 3; 3 |];;

         let _ = transition ex2_3 ex2;;
         let _ = transition ex2_3 ex2;;
         let _ = transition ex2_3 ex2;; (* On a un cycle de taille 3 *)
         let _ = transition ex2_3 ex2;;

Out[23]: val ex2_3 : int array = [|0; 0; 3; 3|]

Out[23]: - : etat = [|1; 0; 1; 2|]

Out[23]: - : etat = [|2; 0; 3; 2|]

Out[23]: - : etat = [|0; 0; 3; 3|]

Out[23]: - : etat = [|1; 0; 1; 2|]
```

2.5 Une autre approche avec des chaînes de Markov

Voici une troisième modélisation, avec des matrices et des chaînes de Markov.

L'idée de base vient de l'observation suivante : **ajouter de l'aléa dans les déplacements des robots devraient permettre de s'assurer (avec une certaine probabilité) que tous les rendez-vous sont bien effectués.**

2.5.1 Échantillonage pondéré

Le module `Random` va être utile.

En Python, on a `numpy.random.choice` pour faire cet échantillonage pondéré, pas en Caml, donc on va l'écrire manuellement.

```
In [24]: Random.init 0;;
```

```
Out[24]: - : unit = ()
```

Etant donné une distribution discrète $\pi = (\pi_1, \dots, \pi_N)$ sur $\{1, \dots, N\}$, la fonction suivante permet de générer un indice i tel que

$$\mathbb{P}(i = k) = \pi_k, \forall k \in \{1, \dots, N\}.$$

```
In [25]: let weight_sampling (pi : float array) () =
             let p = Random.float 1. in
             let i = ref (-1) in
             let acc = ref 0. in
             while !acc < p do
                 incr i;
                 acc := (!acc) +. pi.(!i);
             done;
             !i
         ;;
```

```
Out[25]: val weight_sampling : float array -> unit -> int = <fun>
```

Par exemple, tirer 100 échantillons suivant la distribution $\pi = [0.5, 0.1, 0.4]$ devrait donner environ 50 fois 0, 10 fois 1 et 40 fois 2 :

```
In [26]: let compte (a : 'a array) (x : 'a) : int =
             Array.fold_left (fun i y -> if y = x then i + 1 else i) 0 a
         ;;

         let echantillons = Array.init 100
             (fun _ -> weight_sampling [| 0.5; 0.1; 0.4 |] ())
         ;;

         compte echantillons 0;;
         compte echantillons 1;;
         compte echantillons 2;;
```

```
Out[26]: val compte : 'a array -> 'a -> int = <fun>
```

```
Out[26]: val echantillons : int array =
           [|0; 0; 2; 2; 0; 0; 0; 0; 0; 0; 0; 2; 1; 2; 0; 2; 1; 2; 1; 2; 2; 1; 2; 0;
             1; 2; 2; 0; 2; 0; 2; 0; 0; 0; 2; 2; 2; 2; 2; 0; 2; 0; 2; 0; 0; 2; 2; 2;
             1; 0; 1; 2; 0; 2; 2; 2; 0; 0; 2; 0; 2; 0; 0; 0; 0; 0; 1; 0; 1; 0; 0; 2;
             2; 2; 2; 2; 2; 1; 0; 0; 2; 2; 2; 2; 0; 0; 0; 0; 1; 2; 0; 1; 0; 0; 0; 0;
             1; 0; 0; 2|]
```

```
Out[26]: - : int = 46
```

```
Out[26]: - : int = 13
```

```
Out[26]: - : int = 41
```

46/100, 13/100 et 41/100, c'est pas trop loin de 0.5, 0.1, 0.4.

2.5.2 Simuler une étape d'une chaîne de Markov ?

On peut utiliser cette fonction pour suivre une transition, aléatoire, sur une chaîne de Markov.

```
In [27]: let markov_1 (a : float array array) (i : int) : int =
             let pi = a.(i) in
             weight_sampling pi ()
         ;;
```

```
Out[27]: val markov_1 : float array array -> int -> int = <fun>
```

Avec un petit exemple définit, on peut voir le résultat de 100 transitions différentes depuis l'état 0 :

```
In [28]: let a = [|
                     [| 0.4; 0.3; 0.3 |];
                     [| 0.3; 0.4; 0.3 |];
                     [| 0.3; 0.3; 0.4 |]
             |]
         ;;
```

```
Out[28]: val a : float array array =
           [|[|0.4; 0.3; 0.3|]; [|0.3; 0.4; 0.3|]; [|0.3; 0.3; 0.4|]|]
```

```
In [29]: print "\n";;
         for _ = 0 to 100 do
             print "%i" (markov_1 a 0);
         done;;
         flush_all ();;
```

```
Out[29]: - : unit = ()
```

```
Out[29]: - : unit = ()
```

```
Out[29]: - : unit = ()
```

On peut suivre plusieurs transitions :

```
In [30]: let markov_n (a : float array array) (etat : int) (n : int) : int =
             let u = ref etat in
             for _ = 0 to n-1 do
                 u := markov_1 a !u;
             done;
             !u
         ;;
```

```
Out[30]: val markov_n : float array array -> int -> int -> int = <fun>
```

```
In [31]: markov_n a 0 10;;
```

```
Out[31]: - : int = 2
```

Et pour plusieurs robots, c'est pareil : chaque robots a un état (`robots.(i)`) et une matrice de transition (`a.(i)`) :

```
In [32]: let markovs_n (a : float array array array) (robots : int array) (n : int) : int array
             Array.mapi (fun i u -> markov_n a.(i) u n) robots
         ;;
```

```
Out[32]: val markovs_n : float array array array -> int array -> int -> int array =
           <fun>
```

Si par exemple chaque état a la même matrice de transition :

```
In [33]: markovs_n [|a; a; a|] [|0; 1; 2|] 10;;
```

```
Out[33]: - : int array = [|1; 1; 0|]
```

2.5.3 Modéliser nos robots avec des chaînes de Markov

Plutôt que d'imposer à chaque robot un ordre fixe de ses rendez-vous, on va leur donner une probabilité uniforme d'aller, après un rendez-vous, à n'importe lequel de leur rendez-vous.

- Cela demande de transformer la liste $T_1, \ldots, T_n$ de rendez-vous en n matrices de transition de chaînes de Markov, une par robot.
- Et ensuite de simuler chaque chaîne de Markov, parant d'un état initial $T_i[0]$.

In [34]:
```
(* Fonctions utiles *)
Array.init;;
Array.make_matrix;;
Array.iter;;
```

Out[34]:
```
- : int -> (int -> 'a) -> 'a array = <fun>
```

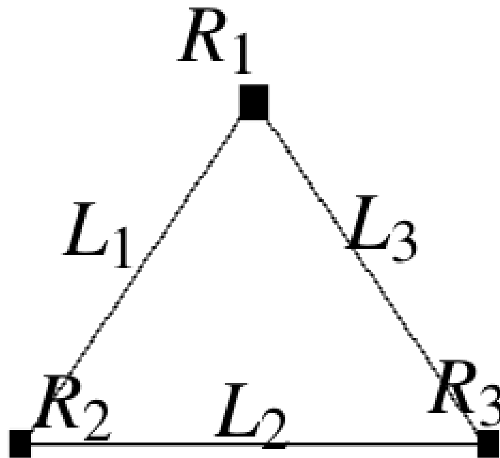
Out[34]:
```
- : int -> int -> 'a -> 'a array array = <fun>
```

Out[34]:
```
- : ('a -> unit) -> 'a array -> unit = <fun>
```

In [35]:
```
let mat_proba_depuis_rdv (ts : int array array) : (float array array) array =
    let n = Array.length ts in
    let a = Array.init n (fun _ -> Array.make_matrix n n 0.) in
    for i = 0 to n-1 do
        (* Pour le robot R_i, ses rendez-vous T_i sont ts.(i) *)
        let r = ts.(i) in
        let m = Array.length r in
        let p_i = 1. /. (float_of_int m) in
        (* Pour chaque rendez-vous L_j dans T_i,
           remplir a.(j).(k) par 1/m,
           et aussi a.(k).(j) par 1/m
           pour chaque autre k dans L_j
        *)
        for j = 0 to m-1 do
            for k = 0 to m-1 do
                a.(i).(r.(j)).(r.(k)) <- p_i;
                a.(i).(r.(k)).(r.(j)) <- p_i;
            done;
        done;
    done;
    a
;;
```

Out[35]:
```
val mat_proba_depuis_rdv : int array array -> float array array array = <fun>
```

Avec les fonctions précédentes, on peut faire évoluer le système.



Premier exemple de robots

```
In [36]: let simule_markov_robots (ts : int array array)
             (etats : int array) (n : int)
             : int array =
             let a = mat_proba_depuis_rdv ts in
             markovs_n a etats n
         ;;
```

```
Out[36]: val simule_markov_robots : int array array -> int array -> int -> int array =
           <fun>
```

2.5.4 Exemple 1

On commence avec le premier exemple du texte, avec trois robots R_1, R_2, R_3, qui ont comme tableaux de rendez-vous $T_1 = [1,3]$, $T_2 = [2,1]$ et $T_3 = [3,2]$.

Ce système n'est pas bloqué, mais aucun rendez-vous n'est réalisé avec l'approche statique. L'approche probabiliste permettra, espérons, de résoudre ce problème.

```
In [37]: let ex3 = [| [|0; 2|]; [|1; 0|]; [|2; 1|]|] ;;
```

```
Out[37]: val ex3 : int array array = [|[|0; 2|]; [|1; 0|]; [|2; 1|]|]
```

On peut écrire une fonction qui récupère l'état initial dans lequel se trouve chaque robot (le texte donnait comme convention d'utiliser le premier de chaque liste).

```
In [38]: let premier_etat (rdvs : int array array) : int array =
             Array.init (Array.length rdvs) (fun i -> rdvs.(i).(0))
         ;;
```

```
Out[38]: val premier_etat : int array array -> int array = <fun>
```

```
In [39]: let ex3_1 = premier_etat ex3;;

Out[39]: val ex3_1 : int array = [|0; 1; 2|]
```

On vérifie la matrice de transition produite par `mat_proba_depuis_rdv` :

```
In [40]: mat_proba_depuis_rdv ex3;;

Out[40]: - : float array array array =
         [|[|[|0.5; 0.; 0.5|]; [|0.; 0.; 0.|]; [|0.5; 0.; 0.5|]|];
           [|[|0.5; 0.5; 0.|]; [|0.5; 0.5; 0.|]; [|0.; 0.; 0.|]|];
           [|[|0.; 0.; 0.|]; [|0.; 0.5; 0.5|]; [|0.; 0.5; 0.5|]|]|]
```

On peut vérifier que ces chaînes de Markov représentent bien le comportement des robots, par exemple le premier robot R_1 a $T_1 = [1, 3]$, donc il alterne entre l'état 0 et 2, avec la matrice de transition

$$\mathbf{A}_i := \begin{bmatrix} 1/2 & 0 & 1/2 \\ 0 & 0 & 0 \\ 1/2 & 0 & 1/2 \end{bmatrix}.$$

Et enfin on peut simuler le système, par exemple pour juste une étape, plusieurs fois (pour bien visualiser).

```
In [41]: simule_markov_robots ex3 ex3_1 0;;  (* rien à faire ! *)

Out[41]: - : int array = [|0; 1; 2|]
```

```
In [42]: simule_markov_robots ex3 ex3_1 1;;

Out[42]: - : int array = [|0; 0; 1|]
```

Pour mieux comprendre le fonctionnement, on va afficher les états intermédiaires.

```
In [43]: let print = Printf.printf;;

Out[43]: val print : ('a, out_channel, unit) format -> 'a = <fun>
```

```
In [44]: let affiche_etat (etat : int array) =
             Array.iter (fun u -> print "%i " u) etat;
             print "\n";
             flush_all ();
         ;;

Out[44]: val affiche_etat : int array -> unit = <fun>
```

```
In [45]: affiche_etat ex3_1;;
```

```
Out[45]: - : unit = ()
```

```
212221110001100111112201200122012120202121011222102100212012000210220020221011200202100201211101
```

```
In [46]: let u = ref ex3_1 in
         for _ = 0 to 10 do
             affiche_etat !u;
             u := simule_markov_robots ex3 !u 1;
         done;;
```

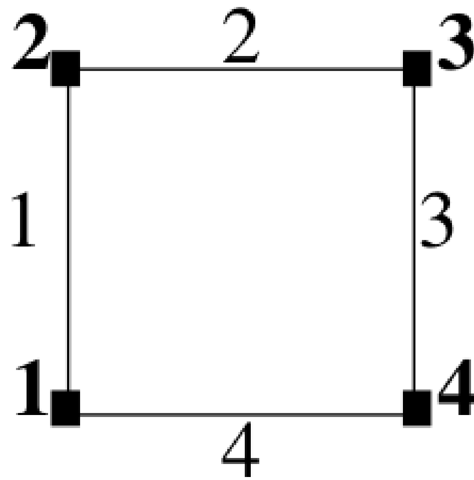
```
0 1 2
2 1 2
0 1 1
0 0 2
0 1 1
0 0 1
0 1 1
0 1 1
2 1 1
2 0 1
0 1 1
```

```
Out[46]: - : unit = ()
```

On constate que des rendez-vous ont bien été effectués ! Pas à chaque fois, mais presque. En tout cas, ça fonctionne mieux que l'approche naïve, peu importe l'état initial.

```
In [47]: let u = ref [| 0; 1; 1 |] in
         for _ = 0 to 10 do
             affiche_etat !u;
             u := simule_markov_robots ex3 !u 1;
         done;;
```

```
0 1 1
0 1 2
0 0 1
0 0 2
2 1 2
0 0 1
2 1 2
2 0 1
2 1 1
0 1 1
0 0 1
```



Premier exemple de robots

```
Out[47]: - : unit = ()
```

Sur 11 états, le rendez-vous 0 a été fait 1 fois, le 1 a été fait 4 fois, et le 2 a été fait 4 fois aussi. Soit 9 sur 11 étapes utiles ! Pas mal !

(ce texte n'est pas valide à chaque exécution à cause de l'aléa...)

2.5.5 Exemple 2

Puis le second exemple :

```
In [48]: let ex4 = [| [|0; 3|]; [|0; 1|]; [|1; 2|]; [|2; 3|] |];;

Out[48]: val ex4 : int array array = [|[|0; 3|]; [|0; 1|]; [|1; 2|]; [|2; 3|]|]
```

```
In [49]: let ex4_1 = premier_etat ex4;;

Out[49]: val ex4_1 : int array = [|0; 0; 1; 2|]
```

On vérifie la matrice de transition produite par `mat_proba_depuis_rdv` :

```
In [50]: mat_proba_depuis_rdv ex4;;

Out[50]: - : float array array array =
         [|[|[|0.5; 0.; 0.; 0.5|]; [|0.; 0.; 0.; 0.|]; [|0.; 0.; 0.; 0.|];
             [|0.5; 0.; 0.; 0.5|]|];
           [|[|0.5; 0.5; 0.; 0.|]; [|0.5; 0.5; 0.; 0.|]; [|0.; 0.; 0.; 0.|];
             [|0.; 0.; 0.; 0.|]|];
```

```
[|[|0.; 0.; 0.; 0.|]; [|0.; 0.5; 0.5; 0.|]; [|0.; 0.5; 0.5; 0.|];
  [|0.; 0.; 0.; 0.|]|];
[|[|0.; 0.; 0.; 0.|]; [|0.; 0.; 0.; 0.|]; [|0.; 0.; 0.5; 0.5|];
  [|0.; 0.; 0.5; 0.5|]|]|]
```

Et enfin on peut simuler le système, par exemple pour juste une étape, plusieurs fois (pour bien visualiser).

```
In [51]: simule_markov_robots ex4 ex4_1 0;;  (* rien à faire ! *)

Out[51]: - : int array = [|0; 0; 1; 2|]
```

```
In [52]: simule_markov_robots ex4 ex4_1 1;;

Out[52]: - : int array = [|3; 1; 1; 2|]
```

```
In [53]: let u = ref ex4_1 in
         for _ = 0 to 10 do
             affiche_etat !u;
             u := simule_markov_robots ex4 !u 1;
         done;;

0 0 1 2
0 0 1 2
0 0 1 3
3 0 1 3
0 0 1 3
0 1 1 2
0 1 2 3
3 0 2 3
0 1 2 3
0 0 2 2
3 0 1 2


Out[53]: - : unit = ()
```

Sur 11 états, le rendez-vous 0 a été fait 5 fois, le 1 a été fait 2 fois, le 2 a été fait 3 fois, et le 3 a été fait 3 fois aussi. Soit plus d'un rendez-vous par étape réussi en moyenne, et tous les rendez-vous ont été vus.

(ce texte n'est pas valide à chaque exécution à cause de l'aléa...)

2.5.6 Conclusion de cette approche par des chaînes de Markov

On ne va pas en faire plus, mais cela suffit de montrer la pertinence de cette autre approche.

Merci d'avoir lu jusque là !

2.6 Conclusion

Voilà pour la question obligatoire de programmation :

- on a adopté la modélisation du texte (dans la 1ère approche) et aussi une autre modélisation (2ème approche),
- on a traité l'exemple du texte, et deux exemples assez proches.
- on a fait des exemples et *on les garde* dans ce qu'on présente au jury.
- on a essayé de faire *un peu plus*, en suivant une des directions suggérées par le texte.

  Bien-sûr, ce petit notebook ne se prétend pas être une solution optimale, ni exhaustive.

N'hésitez pas à aller voir ce dépôt GitHub pour d'autres notebooks, notamment cette page pour d'autres notebooks corrigeant des textes de modélisation (option D, informatique théorique) pour l'agrégation de mathématiques.