Eclairage_graphe

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, 2016-17
1.2 À propos de ce document
1.3 Question de programmation
1.3.1 Exercice
1.3.2 Besoin de plusieurs exemples
1.4 Solution
1.4.1 Représentation du graphe et de la proposition d'éclairage
1.4.2 Vérification
1.4.3 Exemples
1.4.3.1 Pour des éclairages valides.
1.4.3.2 Pour des éclairages non valides :
1.4.4 Bonus : vérifier que le graphe est valide
1.5 Bonus : énumération de tous les éclairages possibles
1.5.1 Énumération des éclairages possibles
1.5.2 Liste de taille minimale parmi une liste de listes
1.5.3 Trouver un éclairage optimal
1.5.4 Exemples
1.6 Complexités en temps et espace (bonus)
1.6.1 En temps
1.6.2 En espace
1.7 Conclusion
1.7.1 Qualités
1.7.2 Défauts
1.7.3 Ouverture ?

2 Texte d'oral de modélisation - Agrégation Option Informatique

2.1 Préparation à l'agrégation - ENS de Rennes, 2016-17

- *Date* : 22 mai 2017
- *Auteur* : Lilian Besson
- *Texte*: Annale 2012, "Éclairage graphe" (public2012-D1)

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]: Sys.command "ocaml -version";;
```

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

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

2.3 Question de programmation

La question de programmation pour ce texte était donnée au tout début, à la page 2 :

2.3.1 Exercice

> Dans le langage de votre choix, implémenter un programme qui étant donnés un graphe et une proposition d'éclairage des lampadaires teste si celle-ci est correcte, *i.e.*, si toutes les rues sont bien éclairées.
>
> Le tester sur différents exemples bien choisis (on pourra justifier la/les structures de données utilisée).

2.3.2 Besoin de plusieurs exemples

Pour une fois, on voit bien que le jury *exige* de tester la fonction sur *plusieurs* exemples.

2.4 Solution

2.4.1 Représentation du graphe et de la proposition d'éclairage

Les graphes ici seront constitués de *places* (= sommets), reliés entre elles par des *rues* (= arêtes).

Pour vérifier un éclairage, donné sous forme d'une liste de places, on va devoir vérifier que chaque rue est connectée à une place éclairée.

Une approche très simple, en trois étapes :

1. compter le nombre de rues,
2. pour chaque place éclairée, compter le nombre de rues qui en partent (et qui sont donc éclairées),
3. s'il y a (au moins) une rue non éclairée, renvoyer `false`, sinon renvoyer `true`.

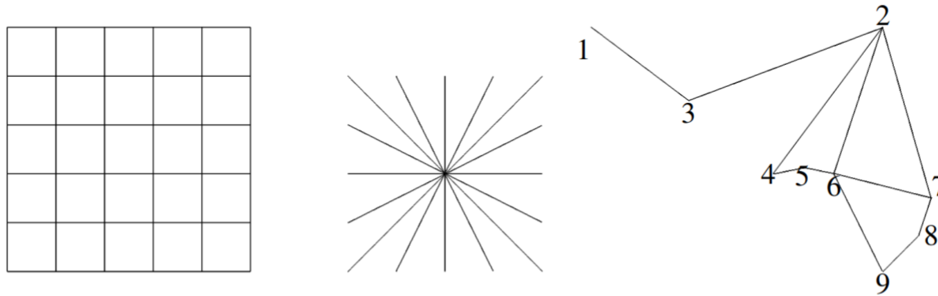

FIG. 1. trois exemples de ville

Graphes de la Figure 1 du texte

Pour être efficace, il faut pouvoir accéder efficacement aux rues qui partent de chaque place (et il suffit de les compter, si on sait qu'elles sont uniques dans la représentation).

$\implies$ À partir de ce constat, on opte pour une représentation du graphe par **liste d'adjacence**.

- Comme les graphes sont non orientés, chaque arête est présente deux fois dans la représentation du graphe : l'arête $u \leftrightarrow v$ est présente comme $v \in V[u]$ (les places voisines de u), et $u \in V[v]$ (les places voisines de v).
- Le fait que les graphes soient planaires n'apportent rien à la représentation.
- Par simplicité, on va représenter les places par leur numéros (on pourra aussi prendre des chaînes de caractères, comme `"Place de l'étoile"`, `"Nation"`, `"Champd de Mars"` etc, mais c'est plus long à écrire).

```
In [2]: type place = int;;
        type rues = place list;;

Out[2]: type place = int


Out[2]: type rues = place list


In [3]: type ville = rues list;;

Out[3]: type ville = rues list
```

On donne tout de suite un exemple de graphe, en prenant le 3ème exemple de la Figure 1 du texte.

```
In [4]: let graphe1 : ville = [
            [2];            (* place 0 *)
            [2; 3; 6];      (* place 1 *)
            [0; 1];         (* place 2 *)
            [1; 4];         (* place 3 *)
```

```
        [3; 5];         (* place 4 *)
        [1; 4; 6; 8];   (* place 5 *)
        [1; 5; 7];      (* place 6 *)
        [6; 8];         (* place 7 *)
        [5; 7];         (* place 8 *)
    ];;
```

```
Out[4]: val graphe1 : ville =
          [[2]; [2; 3; 6]; [0; 1]; [1; 4]; [3; 5]; [1; 4; 6; 8]; [1; 5; 7]; [6; 8];
           [5; 7]]
```

On a besoin que les indices commence à 0 et jusqu'à $n-1$ (où n est le nombre de places), puisque la liste des rues utilisent implicitement la numérotation des places.

```
In [5]: type eclairage = place list;;
```

```
Out[5]: type eclairage = place list
```

Trois exemples d'éclairages, deux satisfaisant donc l'un trivialement, et l'autre non satisfaisant :

```
In [6]: let eclairage1_sat : eclairage = [
            0; 1; 2; 3; 4; 5; 6; 7; 8
        ];;

        let eclairage2_sat : eclairage = [
            1; 2; 3; 5; 6; 8
        ];;
```

```
Out[6]: val eclairage1_sat : eclairage = [0; 1; 2; 3; 4; 5; 6; 7; 8]
```

```
Out[6]: val eclairage2_sat : eclairage = [1; 2; 3; 5; 6; 8]
```

```
In [7]: let eclairage1_nonsat : eclairage = [
            2; 4; 8
        ];;

        let eclairage2_nonsat : eclairage = [
            1; 2; 3; 5; 6
        ];;
```

```
Out[7]: val eclairage1_nonsat : eclairage = [2; 4; 8]
```

```
Out[7]: val eclairage2_nonsat : eclairage = [1; 2; 3; 5; 6]
```

2.4.2 Vérification

On pourrait écrire une première fonction pour vérifier que le graphe est bien valide, selon la représentation décrite ci-dessus, en vérifiant : - que les places forment bien l'ensemble $\{0, \dots, n-1\}$, - que chaque rue n'est présente qu'une fois dans les listes d'adjacences $V[u]$, - qu'aucune place inconnue n'est dans une liste $V[u]$, - et que le graphe est bien symétrique non-orienté, i.e., que $u \in V[u] \Leftrightarrow v \in V[u]$.

... Mais le sujet n'exige rien de tout ça, donc on passe directement à la question demandée.

Quelques fonctions utiles :

```
In [8]: let somme =
            List.fold_left (+) 0
        ;;
```

```
Out[8]: val somme : int list -> int = <fun>
```

On met en place, facilement, l'algorithme décrit plus haut. Une approche très simple, en trois étapes :

1. compter le nombre de rues,
2. pour chaque place éclairée, compter le nombre de rues qui en partent (et qui sont donc éclairées),
3. s'il y a (au moins) une rue non éclairée, renvoyer `false`, sinon renvoyer `true`.

Il y a une finesse : en comptant une fois les rues dont les deux côtés sont éclairés, et deux fois les rues dont un seul côté est éclairé, on peut comparer au nombre de rues comptées doubles.

C'est légérement sous-optimal, comme on va devoir vérifier (en temps linéaire en taille la de l'éclairage) si chaque rue a un ou deux côté éclairés. Mais on gagne en mémoire puisqu'on n'a pas à construire une représentation de toutes les rues.

```
In [9]: let compte_simple_ou_double_0 (ici : place) (voisines : rues) (proposition : eclairage)
            somme
            (List.map (fun voisine ->
                if List.mem voisine proposition
                then 1
                else 2
            ) voisines)
        ;;
```

```
File "[9]", line 1, characters 31-34:
Warning 27: unused variable ici.
```

```
Out[9]: val compte_simple_ou_double_0 : place -> rues -> eclairage -> int = <fun>
```

```
In [10]: let verifie_eclairage_0 (graphe : ville) (proposition : eclairage) : bool =
             (* 1. compter le nombre de rues *)
             let nombre_rues =
```

```
        List.fold_left (fun a b -> a + (List.length b)) 0 (graphe)
        (* (List.length (List.flatten graphe)) *)
    in
    (* 2. pour chaque place *éclairée*, compter le nombre de rues qui en partent *)
    let nombre_rues_eclairees =
        somme
        (List.map (fun place_eclairee ->
            compte_simple_ou_double_0
            place_eclairee
            (List.nth graphe place_eclairee)
            proposition
        ) proposition)
    in
    (* 3. s'il y a (au moins) une rue non éclairée, renvoyer [false], sinon renvoyer
    nombre_rues <= nombre_rues_eclairees
;;
```

```
Out[10]: val verifie_eclairage_0 : ville -> eclairage -> bool = <fun>
```

Notez qu'on peut même faire mieux, puisque cet appel à `List.mem voisine proposition` est inefficace (au pire il est en $\mathcal{O}(n)$, et on le fait au pire $n-1$ fois).

En effet, on peut précalculer, dans la fonction `verifie_eclairage` un *tableau*, de taille n fixée (et connue à l'avance), qui contient en indice i le résultat de `List.mem i proposition` : sauf qu'au lieu d'appeler plusieurs fois la fonction `List.mem`, on a qu'à parcourir la liste `proposition` une fois, et remplir les cases du tableau.

```
In [14]: let compte_simple_ou_double (_ : place) (voisines : rues) (est_eclairee : bool array)
    somme
    (List.map (fun voisine ->
        if est_eclairee.(voisine)
        then 1
        else 2
    ) voisines)
;;
```

```
Out[14]: val compte_simple_ou_double : place -> rues -> bool array -> int = <fun>
```

Cette fonction précalcule, une seule fois, ce tableau.

```
In [15]: let precalcule_places_eclairees (n : int) (proposition : eclairage) : bool array =
    let resultat = Array.make n false in
    List.iter (fun i ->
        resultat.(i) <- true
    ) proposition;
    resultat
;;
```

```
Out[15]: val precalcule_places_eclairees : int -> eclairage -> bool array = <fun>
```

Et enfin, on s'en sert pour la fonction `verifie_eclairage` plus rapide.

```
In [16]: let verifie_eclairage (graphe : ville) (proposition : eclairage) : bool =
             (* 0. précalcul *)
             let n = List.length graphe in
             let est_eclairee = precalcule_places_eclairees n proposition in
             (* 1. compter le nombre de rues *)
             let nombre_rues =
                 List.fold_left (fun a b -> a + (List.length b)) 0 (graphe)
                 (* (List.length (List.flatten graphe)) *)
             in
             (* 2. pour chaque place *éclairée*, compter le nombre de rues qui en partent *)
             let nombre_rues_eclairees =
                 somme
                 (List.map (fun place_eclairee ->
                     compte_simple_ou_double
                     place_eclairee
                     (List.nth graphe place_eclairee)
                     est_eclairee
                 ) proposition)
             in
             (* 3. s'il y a (au moins) une rue non éclairée, renvoyer [false], sinon renvoyer
             nombre_rues <= nombre_rues_eclairees
         ;;

Out[16]: val verifie_eclairage : ville -> eclairage -> bool = <fun>
```

2.4.3 Exemples

Avec le graphe définit plus haut, et les deux propositions d'éclairages :

```
In [17]: graphe1;;

Out[17]: - : ville =
         [[2]; [2; 3; 6]; [0; 1]; [1; 4]; [3; 5]; [1; 4; 6; 8]; [1; 5; 7]; [6; 8];
          [5; 7]]
```

Pour des éclairages valides.

- Le premier est trivialement valide, on éclaire toutes les places donc toutes les rues sont bien éclairées.

```
In [18]: eclairage1_sat;;
         List.map (fun place_eclairee -> List.nth graphe1 place_eclairee) eclairage1_sat;;

         verifie_eclairage graphe1 eclairage1_sat;;     (* true *)
```

```
Out[18]: - : eclairage = [0; 1; 2; 3; 4; 5; 6; 7; 8]
```

```
Out[18]: - : rues list =
         [[2]; [2; 3; 6]; [0; 1]; [1; 4]; [3; 5]; [1; 4; 6; 8]; [1; 5; 7]; [6; 8];
          [5; 7]]
```

```
Out[18]: - : bool = true
```

- Le second est moins trivial, mais en vérifiant à la main sur le graphe on voit qu'il fonctionne.
  - On a juste éteint la place 0, mais l'arête $0-1$ reste éclairée par la place 1,
  - la place 4 mais les arêtes $3-4$ et $4-5$ sont éclairées par les places 3 et 5,
  - et la place 7 mais les arêtes $6-7$ et $7-8$ sont éclairées par les places 6 et 8.

```
In [19]: eclairage2_sat;;
         List.map (fun place_eclairee -> List.nth graphe1 place_eclairee) eclairage2_sat;;

         verifie_eclairage graphe1 eclairage2_sat;;    (* true *)
```

```
Out[19]: - : eclairage = [1; 2; 3; 5; 6; 8]
```

```
Out[19]: - : rues list = [[2; 3; 6]; [0; 1]; [1; 4]; [1; 4; 6; 8]; [1; 5; 7]; [5; 7]]
```

```
Out[19]: - : bool = true
```

- On peut essayer un éclairage de seulement 5 villes. Est-ce optimal ? (on répondra plus tard, mais oui)

```
In [20]: let eclairage3_sat : eclairage = [
             2; 3; 5; 6; 7;
         ];;

         List.map (fun place_eclairee -> List.nth graphe1 place_eclairee) eclairage3_sat;;

         verifie_eclairage graphe1 eclairage3_sat;;    (* true *)
```

```
Out[20]: val eclairage3_sat : eclairage = [2; 3; 5; 6; 7]
```

```
Out[20]: - : rues list = [[0; 1]; [1; 4]; [1; 4; 6; 8]; [1; 5; 7]; [6; 8]]
```

```
Out[20]: - : bool = true
```

Pour des éclairages non valides :

- Clairement, le premier est trop gourmand et on a éteint trop de place.

```
In [21]: eclairage1_nonsat;;
         List.map (fun place_eclairee -> List.nth graphe1 place_eclairee) eclairage1_nonsat;;

         verifie_eclairage graphe1 eclairage1_nonsat;;   (* false *)

Out[21]: - : eclairage = [2; 4; 8]


Out[21]: - : rues list = [[0; 1]; [3; 5]; [5; 7]]


Out[21]: - : bool = false
```

- Le second semble bon, mais la rue $8-9$ par exemple est éteinte.

```
In [22]: eclairage2_nonsat;;
         List.map (fun place_eclairee -> List.nth graphe1 place_eclairee) eclairage2_nonsat;;

         verifie_eclairage graphe1 eclairage2_nonsat;;   (* false *)

Out[22]: - : eclairage = [1; 2; 3; 5; 6]


Out[22]: - : rues list = [[2; 3; 6]; [0; 1]; [1; 4]; [1; 4; 6; 8]; [1; 5; 7]]


Out[22]: - : bool = false
```

2.4.4 Bonus : vérifier que le graphe est valide

Bon… par acquis de conscience, on implémente l'étape de vérification évoquée plus haut.
D'abord, quelques fonctions utiles :

```
In [23]: let max_liste (liste : int list) : int =
             List.fold_left max (-max_int) liste
         ;;

Out[23]: val max_liste : int list -> int = <fun>


In [24]: max_liste [123; 12; 1];;

Out[24]: - : int = 123
```

```
In [25]: let max_liste_liste (listeliste : int list list) : int =
             max_liste (List.map max_liste listeliste)
         ;;
```

```
Out[25]: val max_liste_liste : int list list -> int = <fun>
```

```
In [26]: max_liste_liste [[123; 12; 1]; [1234; 13]];;
```

```
Out[26]: - : int = 1234
```

```
In [27]: List.for_all (fun x -> (0 <= x) && (x <= 1234)) (List.flatten [[123; 12; 1]; [1234; 13
```

```
Out[27]: - : bool = true
```

```
In [28]: let compte_occurences (liste : int list) (x : int) =
             let rec aux xs x acc =
                 match xs with
                 | [] -> acc
                 | y :: ys when y = x -> aux ys x (acc + 1)
                 | _ :: ys -> aux ys x acc
             in
             aux liste x 0
         ;;
```

```
Out[28]: val compte_occurences : int list -> int -> int = <fun>
```

```
In [29]: compte_occurences [1; 2; 3; 4; 5] 1;;
         compte_occurences [1; 2; 3; 4; 5] 10;;
         compte_occurences [1; 2; 3; 4; 5; 1; 6; 7] 1;;
```

```
Out[29]: - : int = 1
```

```
Out[29]: - : int = 0
```

```
Out[29]: - : int = 2
```

Puis, la vérification annoncée :

```
In [30]: let graphe_valide (graphe : ville) : bool =
             let n = max_liste_liste graphe in
             (* D'abord, on vérifie qu'il y a exactement n + 1 listes de places voisines *)
             let test1 =
                 (List.length graphe) = (n + 1)
```

```
        in
        (* Ensuite, on vérifie que toutes les places voisines sont bien dans des places v
        let test2 =
            List.for_all (
                fun x -> (0 <= x) && (x <= n)
            ) (List.flatten graphe)
        in
        (* Enfin, on vérifie qu'une place voisine v n'est présente qu'une fois dans V[u]
        let test3 =
            List.for_all (
                fun voisines -> List.for_all (
                    fun place ->
                        (compte_occurences voisines place) = 1
                ) voisines
            ) graphe
        in
        (* test1, test2, test3 *)
        test1 && test2 && test3
    ;;
```

Out[30]: val graphe_valide : ville -> bool = <fun>

In [31]:
```
graphe1;;
```

Out[31]:
```
- : ville =
[[2]; [2; 3; 6]; [0; 1]; [1; 4]; [3; 5]; [1; 4; 6; 8]; [1; 5; 7]; [6; 8];
 [5; 7]]
```

In [32]:
```
graphe_valide graphe1;;
```

Out[32]: - : bool = true

On doit aussi tester trois cas de graphes qui contredisent un de chaque test :

- S'il manque une place dans la liste d'adjacence :

In [33]:
```
let graphe2 : ville = [
    [2];            (* place 0 *)
    [2; 3; 6];      (* place 1 *)
    [0; 1];         (* place 2 *)
    [1; 3];         (* place 3 *)
    [3; 5];         (* place 4 *)
    [1; 4; 6; 8];   (* place 5 *)
    [1; 5; 7];      (* place 6 *)
    [6; 8];         (* place 7 *)
                    (* place 8 absente ! *)
];;
```

```
Out[33]: val graphe2 : ville =
           [[2]; [2; 3; 6]; [0; 1]; [1; 3]; [3; 5]; [1; 4; 6; 8]; [1; 5; 7]; [6; 8]]
```

```
In [34]: graphe_valide graphe2;;
```

```
Out[34]: - : bool = false
```

- Si la liste d'adjacence est trop grande :

```
In [35]: let graphe3 : ville = [
             [2];            (* place 0 *)
             [2; 3; 6];      (* place 1 *)
             [0; 1];         (* place 2 *)
             [1; 3];         (* place 3 *)
             [3; 5];         (* place 4 *)
             [1; 4; 6; 8];   (* place 5 *)
             [1; 5; 7];      (* place 6 *)
             [6; 8];         (* place 7 *)
             [5; 7];         (* place 8 absente ! *)
             [5; 7]          (* place 9 ?! *)
         ];;
```

```
Out[35]: val graphe3 : ville =
           [[2]; [2; 3; 6]; [0; 1]; [1; 3]; [3; 5]; [1; 4; 6; 8]; [1; 5; 7]; [6; 8];
            [5; 7]; [5; 7]]
```

```
In [36]: graphe_valide graphe3;;
```

```
Out[36]: - : bool = false
```

- Si une des places voisines n'est pas valide :

```
In [37]: let graphe4 : ville = [
             [2];            (* place 0 *)
             [2; 3; 6];      (* place 1 *)
             [0; 1];         (* place 2 *)
             [1; 3];         (* place 3 *)
             [3; 5];         (* place 4 *)
             [1; 4; 6; 8];   (* place 5 *)
             [1; 5; 7];      (* place 6 *)
             [6; 8];         (* place 7 *)
             [50]            (* place 8 -> 50 ?! *)
         ];;
```

```
Out[37]: val graphe4 : ville =
           [[2]; [2; 3; 6]; [0; 1]; [1; 3]; [3; 5]; [1; 4; 6; 8]; [1; 5; 7]; [6; 8];
            [50]]
```

```
In [38]: graphe_valide graphe4;;
```

```
Out[38]: - : bool = false
```

- Et enfin, si une des listes d'adjacence contient deux fois la même place voisine :

```
In [39]: let graphe5 : ville = [
            [2];            (* place 0 *)
            [2; 3; 6];      (* place 1 *)
            [0; 1];         (* place 2 *)
            [1; 3];         (* place 3 *)
            [3; 5];         (* place 4 *)
            [1; 4; 6; 8];   (* place 5 *)
            [1; 5; 7];      (* place 6 *)
            [6; 8];         (* place 7 *)
            [5; 7; 5]       (* place 8 -> 5 deux fois ! *)
        ];;
```

```
Out[39]: val graphe5 : ville =
           [[2]; [2; 3; 6]; [0; 1]; [1; 3]; [3; 5]; [1; 4; 6; 8]; [1; 5; 7]; [6; 8];
            [5; 7; 5]]
```

```
In [40]: graphe_valide graphe5;;
```

```
Out[40]: - : bool = false
```

2.5 Bonus : énumération de tous les éclairages possibles

Pour de tous petits graphes, on peut suivre l'approche naïve qui consiste à énumérer toutes les possibilités, et renvoyer celle de nombre de lampadaires minimal.

On pourra ensuite vérifier avec la fonction précédente que l'éclairage donné est bien valide.

2.5.1 Énumération des éclairages possibles

Un éclairage est un sous-ensemble, potentiellement vide, de $\{0, \dots, n-1\}$. Il y en a 2^n en tout.

Comment les générer efficacement ? En fait, on s'en fiche, le reste du code sera déjà exponentiel en n...

- La fonction suivante prend une liste de liste, `acc`, et la dédouble en ajoutant `x` aux listes qui ne le contiennent pas encore, et en gardant une copie des listes d'avant. En travaillant avec des listes toutes distinctes, la liste `acc` est toujours dédoublée.

```
In [41]: let ajoute_ou_pas (x : 'a) (acc : 'a list list) =
            (List.map (fun liste ->
                if (List.mem x liste)
```

```
            then liste
            else x :: liste
        ) acc
        )
        @
        (List.filter (fun liste ->
            not (List.mem x liste)
        ) acc
        )
;;
```

```
Out[41]: val ajoute_ou_pas : 'a -> 'a list list -> 'a list list = <fun>
```

```
In [42]: ajoute_ou_pas 1 [[]];;
         ajoute_ou_pas 2 [[1]; []];;
```

```
Out[42]: - : int list list = [[1]; []]
```

```
Out[42]: - : int list list = [[2; 1]; [2]; [1]; []]
```

- Maintenant, il suffit d'itérer cette fonction, depuis la liste `[[]]` contenant juste la liste vide.

```
In [43]: let tous_sous_ensembles (liste : 'a list) : 'a list list =
             let rec aux xs acc =
                 match xs with
                 | [] -> acc
                 | y :: ys ->
                     aux ys (ajoute_ou_pas y acc)
             in
             aux liste [ [] ]
         ;;
```

```
Out[43]: val tous_sous_ensembles : 'a list -> 'a list list = <fun>
```

```
In [44]: tous_sous_ensembles [];;
         tous_sous_ensembles [1];;
         tous_sous_ensembles [1; 2];;
         tous_sous_ensembles [1; 2; 3];; (* taille 2^3 = 8 *)
```

```
Out[44]: - : 'a list list = [[]]
```

```
Out[44]: - : int list list = [[1]; []]
```

```
Out[44]: - : int list list = [[2; 1]; [2]; [1]; []]
```

```
Out[44]: - : int list list = [[3; 2; 1]; [3; 2]; [3; 1]; [3]; [2; 1]; [2]; [1]; []]
```

```
In [45]: tous_sous_ensembles [0; 1; 2; 3];; (* taille 2^4 = 16 *)
         tous_sous_ensembles [0; 1; 2; 3; 4];; (* taille 2^5 = 32 *)
         tous_sous_ensembles [0; 1; 2; 3; 4; 5];; (* taille 2^6 = 64 *)
         tous_sous_ensembles [0; 1; 2; 3; 4; 5; 6];; (* taille 2^7 = 128 *)
         tous_sous_ensembles [0; 1; 2; 3; 4; 5; 6; 7];; (* taille 2^8 = 256 *)
         tous_sous_ensembles [0; 1; 2; 3; 4; 5; 6; 7; 8];; (* taille 2^9 = 512 *)

         assert ((List.length (tous_sous_ensembles [0; 1; 2; 3; 4; 5; 6; 7; 8])) = 512);;
```

```
Out[45]: - : int list list =
         [[3; 2; 1; 0]; [3; 2; 1]; [3; 2; 0]; [3; 2]; [3; 1; 0]; [3; 1]; [3; 0];
          [3]; [2; 1; 0]; [2; 1]; [2; 0]; [2]; [1; 0]; [1]; [0]; []]
```

```
Out[45]: - : int list list =
         [[4; 3; 2; 1; 0]; [4; 3; 2; 1]; [4; 3; 2; 0]; [4; 3; 2]; [4; 3; 1; 0];
          [4; 3; 1]; [4; 3; 0]; [4; 3]; [4; 2; 1; 0]; [4; 2; 1]; [4; 2; 0]; [4; 2];
          [4; 1; 0]; [4; 1]; [4; 0]; [4]; [3; 2; 1; 0]; [3; 2; 1]; [3; 2; 0];
          [3; 2]; [3; 1; 0]; [3; 1]; [3; 0]; [3]; [2; 1; 0]; [2; 1]; [2; 0]; [2];
          [1; 0]; [1]; [0]; []]
```

```
Out[45]: - : int list list =
         [[5; 4; 3; 2; 1; 0]; [5; 4; 3; 2; 1]; [5; 4; 3; 2; 0]; [5; 4; 3; 2];
          [5; 4; 3; 1; 0]; [5; 4; 3; 1]; [5; 4; 3; 0]; [5; 4; 3]; [5; 4; 2; 1; 0];
          [5; 4; 2; 1]; [5; 4; 2; 0]; [5; 4; 2]; [5; 4; 1; 0]; [5; 4; 1]; [5; 4; 0];
          [5; 4]; [5; 3; 2; 1; 0]; [5; 3; 2; 1]; [5; 3; 2; 0]; [5; 3; 2];
          [5; 3; 1; 0]; [5; 3; 1]; [5; 3; 0]; [5; 3]; [5; 2; 1; 0]; [5; 2; 1];
          [5; 2; 0]; [5; 2]; [5; 1; 0]; [5; 1]; [5; 0]; [5]; [4; 3; 2; 1; 0];
          [4; 3; 2; 1]; [4; 3; 2; 0]; [4; 3; 2]; [4; 3; 1; 0]; [4; 3; 1]; [4; 3; 0];
          [4; 3]; [4; 2; 1; 0]; [4; 2; 1]; [4; 2; 0]; [4; 2]; [4; 1; 0]; [4; 1];
          [4; 0]; [4]; [3; 2; 1; 0]; [3; 2; 1]; [3; 2; 0]; [3; 2]; [3; 1; 0];
          [3; 1]; [3; 0]; [3]; [2; 1; 0]; [2; 1]; [2; 0]; [2]; [1; 0]; [1]; [0];
          []]
```

```
Out[45]: - : int list list =
         [[6; 5; 4; 3; 2; 1; 0]; [6; 5; 4; 3; 2; 1]; [6; 5; 4; 3; 2; 0];
          [6; 5; 4; 3; 2]; [6; 5; 4; 3; 1; 0]; [6; 5; 4; 3; 1]; [6; 5; 4; 3; 0];
          [6; 5; 4; 3]; [6; 5; 4; 2; 1; 0]; [6; 5; 4; 2; 1]; [6; 5; 4; 2; 0];
          [6; 5; 4; 2]; [6; 5; 4; 1; 0]; [6; 5; 4; 1]; [6; 5; 4; 0]; [6; 5; 4];
          [6; 5; 3; 2; 1; 0]; [6; 5; 3; 2; 1]; [6; 5; 3; 2; 0]; [6; 5; 3; 2];
          [6; 5; 3; 1; 0]; [6; 5; 3; 1]; [6; 5; 3; 0]; [6; 5; 3]; [6; 5; 2; 1; 0];
          [6; 5; 2; 1]; [6; 5; 2; 0]; [6; 5; 2]; [6; 5; 1; 0]; [6; 5; 1]; [6; 5; 0];
          [6; 5]; [6; 4; 3; 2; 1; 0]; [6; 4; 3; 2; 1]; [6; 4; 3; 2; 0]; [6; 4; 3; 2];
          [6; 4; 3; 1; 0]; [6; 4; 3; 1]; [6; 4; 3; 0]; [6; 4; 3]; [6; 4; 2; 1; 0];
```

```
 [6; 4; 2; 1]; [6; 4; 2; 0]; [6; 4; 2]; [6; 4; 1; 0]; [6; 4; 1]; [6; 4; 0];
 [6; 4]; [6; 3; 2; 1; 0]; [6; 3; 2; 1]; [6; 3; 2; 0]; [6; 3; 2];
 [6; 3; 1; 0]; [6; 3; 1]; [6; 3; 0]; [6; 3]; [6; 2; 1; 0]; [6; ...]; ...]
```

```
Out[45]: - : int list list =
         [[7; 6; 5; 4; 3; 2; 1; 0]; [7; 6; 5; 4; 3; 2; 1]; [7; 6; 5; 4; 3; 2; 0];
          [7; 6; 5; 4; 3; 2]; [7; 6; 5; 4; 3; 1; 0]; [7; 6; 5; 4; 3; 1];
          [7; 6; 5; 4; 3; 0]; [7; 6; 5; 4; 3]; [7; 6; 5; 4; 2; 1; 0];
          [7; 6; 5; 4; 2; 1]; [7; 6; 5; 4; 2; 0]; [7; 6; 5; 4; 2]; [7; 6; 5; 4; 1; 0];
          [7; 6; 5; 4; 1]; [7; 6; 5; 4; 0]; [7; 6; 5; 4]; [7; 6; 5; 3; 2; 1; 0];
          [7; 6; 5; 3; 2; 1]; [7; 6; 5; 3; 2; 0]; [7; 6; 5; 3; 2]; [7; 6; 5; 3; 1; 0];
          [7; 6; 5; 3; 1]; [7; 6; 5; 3; 0]; [7; 6; 5; 3]; [7; 6; 5; 2; 1; 0];
          [7; 6; 5; 2; 1]; [7; 6; 5; 2; 0]; [7; 6; 5; 2]; [7; 6; 5; 1; 0];
          [7; 6; 5; 1]; [7; 6; 5; 0]; [7; 6; 5]; [7; 6; 4; 3; 2; 1; 0];
          [7; 6; 4; 3; 2; 1]; [7; 6; 4; 3; 2; 0]; [7; 6; 4; 3; 2]; [7; 6; 4; 3; 1; 0];
          [7; 6; 4; 3; 1]; [7; 6; 4; 3; 0]; [7; 6; 4; 3]; [7; 6; 4; 2; 1; 0];
          [7; 6; 4; 2; 1]; [7; 6; 4; 2; 0]; [7; 6; 4; 2]; [7; 6; 4; 1; 0];
          [7; 6; 4; 1]; [7; 6; 4; ...]; ...]
```

```
Out[45]: - : int list list =
         [[8; 7; 6; 5; 4; 3; 2; 1; 0]; [8; 7; 6; 5; 4; 3; 2; 1];
          [8; 7; 6; 5; 4; 3; 2; 0]; [8; 7; 6; 5; 4; 3; 2]; [8; 7; 6; 5; 4; 3; 1; 0];
          [8; 7; 6; 5; 4; 3; 1]; [8; 7; 6; 5; 4; 3; 0]; [8; 7; 6; 5; 4; 3];
          [8; 7; 6; 5; 4; 2; 1; 0]; [8; 7; 6; 5; 4; 2; 1]; [8; 7; 6; 5; 4; 2; 0];
          [8; 7; 6; 5; 4; 2]; [8; 7; 6; 5; 4; 1; 0]; [8; 7; 6; 5; 4; 1];
          [8; 7; 6; 5; 4; 0]; [8; 7; 6; 5; 4]; [8; 7; 6; 5; 3; 2; 1; 0];
          [8; 7; 6; 5; 3; 2; 1]; [8; 7; 6; 5; 3; 2; 0]; [8; 7; 6; 5; 3; 2];
          [8; 7; 6; 5; 3; 1; 0]; [8; 7; 6; 5; 3; 1]; [8; 7; 6; 5; 3; 0];
          [8; 7; 6; 5; 3]; [8; 7; 6; 5; 2; 1; 0]; [8; 7; 6; 5; 2; 1];
          [8; 7; 6; 5; 2; 0]; [8; 7; 6; 5; 2]; [8; 7; 6; 5; 1; 0]; [8; 7; 6; 5; 1];
          [8; 7; 6; 5; 0]; [8; 7; 6; 5]; [8; 7; 6; 4; 3; 2; 1; 0];
          [8; 7; 6; 4; 3; 2; 1]; [8; 7; 6; 4; 3; 2; 0]; [8; 7; 6; 4; 3; 2];
          [8; 7; 6; 4; 3; 1; 0]; [8; 7; 6; 4; 3; 1]; [8; 7; 6; 4; 3; 0];
          [8; 7; 6; 4; ...]; ...]
```

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

C'est tellement plus simple en Python…

Une autre fonction utile :

```
In [46]: let range (n : int) : (int list) =
             let rec range_x (n : int) : (int list) =
             match n with
             | n when n < 0 -> []
```

```
        | 0 -> [0]
        | n -> n :: (range_x (n - 1))
    in
    List.rev (range_x n)
;;
```

Out[46]: val range : int -> int list = <fun>

In [47]:
```
range 0;;
range 1;;
range 5;;
range 10;;
```

Out[47]: - : int list = [0]

Out[47]: - : int list = [0; 1]

Out[47]: - : int list = [0; 1; 2; 3; 4; 5]

Out[47]: - : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

In [48]:
```
let tous_les_eclairages_possibles (graphe : ville) : (eclairage list) =
    assert (graphe_valide graphe);
    let n = (List.length graphe) - 1 in
    let sommets = range n in
    tous_sous_ensembles sommets
;;
```

Out[48]: val tous_les_eclairages_possibles : ville -> eclairage list = <fun>

In [49]:
```
tous_les_eclairages_possibles graphe1;;
```

Out[49]:
```
- : eclairage list =
[[8; 7; 6; 5; 4; 3; 2; 1; 0]; [8; 7; 6; 5; 4; 3; 2; 1];
 [8; 7; 6; 5; 4; 3; 2; 0]; [8; 7; 6; 5; 4; 3; 2]; [8; 7; 6; 5; 4; 3; 1; 0];
 [8; 7; 6; 5; 4; 3; 1]; [8; 7; 6; 5; 4; 3; 0]; [8; 7; 6; 5; 4; 3];
 [8; 7; 6; 5; 4; 2; 1; 0]; [8; 7; 6; 5; 4; 2; 1]; [8; 7; 6; 5; 4; 2; 0];
 [8; 7; 6; 5; 4; 2]; [8; 7; 6; 5; 4; 1; 0]; [8; 7; 6; 5; 4; 1];
 [8; 7; 6; 5; 4; 0]; [8; 7; 6; 5; 4]; [8; 7; 6; 5; 3; 2; 1; 0];
 [8; 7; 6; 5; 3; 2; 1]; [8; 7; 6; 5; 3; 2; 0]; [8; 7; 6; 5; 3; 2];
 [8; 7; 6; 5; 3; 1; 0]; [8; 7; 6; 5; 3; 1]; [8; 7; 6; 5; 3; 0];
 [8; 7; 6; 5; 3]; [8; 7; 6; 5; 2; 1; 0]; [8; 7; 6; 5; 2; 1];
 [8; 7; 6; 5; 2; 0]; [8; 7; 6; 5; 2]; [8; 7; 6; 5; 1; 0]; [8; 7; 6; 5; 1];
 [8; 7; 6; 5; 0]; [8; 7; 6; 5]; [8; 7; 6; 4; 3; 2; 1; 0];
 [8; 7; 6; 4; 3; 2; 1]; [8; 7; 6; 4; 3; 2; ...]; ...]
```

2.5.2 Liste de taille minimale parmi une liste de listes

```
In [50]: let min_liste (liste : int list) : int =
             List.fold_left min (max_int) liste
         ;;
```

```
Out[50]: val min_liste : int list -> int = <fun>
```

```
In [51]: min_liste [123; 12; 1];;
```

```
Out[51]: - : int = 1
```

```
In [52]: let taille_min (listes : int list list) : int =
             min_liste (List.map List.length listes)
         ;;
```

```
Out[52]: val taille_min : int list list -> int = <fun>
```

2.5.3 Trouver un éclairage optimal

Ca va être assez naïf :

1. on calcule tous les éclairages possibles,
2. on filtre les éclairages pour ne garder que ceux qui sont valides,
3. on calcule la longueur minimale,
4. tant qu'on y est, on filtre pour ne garder que ceux qui sont de longueur minimale (plutôt que d'en renvoyer un seul).

```
In [53]: let eclairages_optimaux (graphe : ville) : (eclairage list) =
             let propositions =
                 tous_les_eclairages_possibles graphe
             in
             let propositions_valides =
                 List.filter (verifie_eclairage graphe) propositions
             in
             let taille_minimale =
                 taille_min propositions_valides
             in
             let propositions_minimales =
                 List.filter (fun li ->
                     (List.length li) = taille_minimale
                 ) propositions_valides
             in
             propositions_minimales
         ;;
```

```
Out[53]: val eclairages_optimaux : ville -> eclairage list = <fun>
```

2.5.4 Exemples

- On commence avec un tout petit graphe, pour vérifier rapidement avant de passer sur un plus gros graphe :

```
In [54]: let graphe2 : ville = [
             [1; 2];   (* place 0 -> 1 2 *)
             [0; 2];   (* place 1 -> 0 2 *)
             [0; 1]    (* place 2 -> 0 1 *)
         ] ;;
```

```
Out[54]: val graphe2 : ville = [[1; 2]; [0; 2]; [0; 1]]
```

```
In [55]: tous_les_eclairages_possibles graphe2;;

         List.length (tous_les_eclairages_possibles graphe2);;
```

```
Out[55]: - : eclairage list = [[2; 1; 0]; [2; 1]; [2; 0]; [2]; [1; 0]; [1]; [0]; []]
```

```
Out[55]: - : int = 8
```

Il y a 8 éclairages possibles (2^3) et les 3 de taille 2 sont optimaux : en effet sur un graphe complet à n sommets, il faut et il suffit d'éclairer n'importe quel ensenble de $n-1$ places pour éclairer toutes les $n(n-1)$ arêtes.

```
In [56]: eclairages_optimaux graphe2;;
```

```
Out[56]: - : eclairage list = [[2; 1]; [2; 0]; [1; 0]]
```

- Et sur le graphe avec lequel on travaille depuis le début :

```
In [57]: graphe1;;
```

```
Out[57]: - : ville =
         [[2]; [2; 3; 6]; [0; 1]; [1; 4]; [3; 5]; [1; 4; 6; 8]; [1; 5; 7]; [6; 8];
          [5; 7]]
```

```
In [58]: tous_les_eclairages_possibles graphe1;;

         List.length (tous_les_eclairages_possibles graphe1);;
```

```
Out[58]: - : eclairage list =
         [[8; 7; 6; 5; 4; 3; 2; 1; 0]; [8; 7; 6; 5; 4; 3; 2; 1];
          [8; 7; 6; 5; 4; 3; 2; 0]; [8; 7; 6; 5; 4; 3; 2]; [8; 7; 6; 5; 4; 3; 1; 0];
          [8; 7; 6; 5; 4; 3; 1]; [8; 7; 6; 5; 4; 3; 0]; [8; 7; 6; 5; 4; 3];
```

```
[8; 7; 6; 5; 4; 2; 1; 0]; [8; 7; 6; 5; 4; 2; 1]; [8; 7; 6; 5; 4; 2; 0];
[8; 7; 6; 5; 4; 2]; [8; 7; 6; 5; 4; 1; 0]; [8; 7; 6; 5; 4; 1];
[8; 7; 6; 5; 4; 0]; [8; 7; 6; 5; 4]; [8; 7; 6; 5; 3; 2; 1; 0];
[8; 7; 6; 5; 3; 2; 1]; [8; 7; 6; 5; 3; 2; 0]; [8; 7; 6; 5; 3; 2];
[8; 7; 6; 5; 3; 1; 0]; [8; 7; 6; 5; 3; 1]; [8; 7; 6; 5; 3; 0];
[8; 7; 6; 5; 3]; [8; 7; 6; 5; 2; 1; 0]; [8; 7; 6; 5; 2; 1];
[8; 7; 6; 5; 2; 0]; [8; 7; 6; 5; 2]; [8; 7; 6; 5; 1; 0]; [8; 7; 6; 5; 1];
[8; 7; 6; 5; 0]; [8; 7; 6; 5]; [8; 7; 6; 4; 3; 2; 1; 0];
[8; 7; 6; 4; 3; 2; 1]; [8; 7; 6; 4; 3; 2; ...]; ...]
```

```
Out[58]: - : int = 512
```

```
In [59]: eclairages_optimaux graphe1;;
```

```
Out[59]: - : eclairage list =
[[8; 6; 5; 3; 2]; [7; 6; 5; 3; 2]; [7; 5; 4; 2; 1]; [7; 5; 4; 1; 0];
 [7; 5; 3; 2; 1]; [7; 5; 3; 1; 0]]
```

On vérifie ici que l'éclairage à 5 places qu'on avait proposé plus haut est bien optimal.

2.6 Complexités en temps et espace (bonus)

Il est toujours utile de préciser, rapidement à l'oral et/ou dans le code (un commentaire suffit) les complexité (ou ordre de grandeur) des fonctions exigées par l'énoncé.

Le*s* complexité*s* ? Oui, il ne faut pas oublier l'espace (trop souvent négligé !).

Si vous n'êtes pas sûr ou ne savez pas comment le justifier, mieux vaut marquer :

ń En $\mathcal{O}(n^2)$ en temps et en espace. ż

que de marquer quelque chose de difficilement justifiable comme :

ń Probablement en $\mathcal{O}(n)$ en temps et en espace. ż

2.6.1 En temps

- La première fonction de vérification proposée `verifie_eclairage_0` est linéaire en temps, en m le nombre de rues et p le nombre de place éclairées, i.e., est en $\mathcal{O}(mp)$. C'est sous-optimal pour vérifier que toutes les rues sont bien éclairées, on devrait réussir à faire mieux en $\mathcal{O}(\max(n, m))$!
- La deuxième fonction de vérification proposée `verifie_eclairage` est elle bien linéaire en temps, seulement en m le nombre de rues, i.e., est en $\mathcal{O}(\max(n, m))$ (s'il y a moins de rues que de places, il faut quand même créer le tableau `est_eclairee` en $\mathcal{O}(n)$). C'est optimal pour vérifier que toutes les rues sont bien éclairées ! Youpiiiiiii!!!!!!!!!!!
- Les deux fonctions `tous_les_eclairages_possibles` et `eclairages_optimaux` sont en $\mathcal{O}(2^n)$ en temps, et c'est *beaucoup* !

2.6.2 En espace (ou en mémoire)

- La première fonction de vérification proposée `verifie_eclairage` est aussi linéaire en espace en la taille du graphe.
- La deuxième fonction de vérification proposée `verifie_eclairage_0` est aussi linéaire en espace en la taille du graphe.
- Les deux fonctions `tous_les_eclairages_possibles` et `eclairages_optimaux` sont aussi en $\mathcal{O}(2^n)$ en mémoire, et c'est *beaucoup* ! A noter par contre que le résultat renvoyé par `eclairages_optimaux` est, lui, de taille bien plus raisonnables, mais toujours possiblement exponentielle (si le nombre minimal de places éclairées est k, alors $\binom{n}{k}$).

2.7 Conclusion

Voilà pour la question obligatoire de programmation, et un gros bonus.

2.7.1 Qualités

- On a décomposé le problème en sous-fonctions,
- on a fait des exemples et *on les garde* dans ce qu'on présente au jury,
- on a testé la fonction exigée sur différents exemples.

2.7.2 Défauts

- La première implémentation proposée de `verifie_eclairage_0` n'est pas optimale,
- La deuxième implémentation n'était pas immédiate,
- et on a pas testé sur des exemples très ambitieux.

2.7.3 Ouverture ?

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