Crime_parfait

September 13, 2017

1 Table of Contents

2 Texte d'oral de modélisation - Agrégation Option Informatique

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

- *Date* : 3 avril 2017
- *Auteur* : Lilian Besson
- *Texte*: Annale 2006, "Crime Parfait"

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 Implémentation

La question d'implémentation était la question 2) en page 7.

> ń Proposer une structure de donnée adaptée pour représenter un graphe d'intervalles dont une représentation sous forme de famille d'intervalles est connue. Implémenter de manière efficace l'algorithme de coloriage de graphes d'intervalles et illustrer cet algorithme sur une application bien choisie citée dans le texte. ż

Nous allons donc d'abord définir une structure de donnée pour une famille d'intervalles ainsi que pour un graphe d'intervalle, ainsi qu'une fonction convertissant l'un en l'autre.

Cela permettra de facilement définr les différents exemples du texte, et de les résoudre.

2.3.1 Une bonne structure de donnée pour des intervalles et des graphes d'intervales

- Pour des **intervalles** à valeurs réelles, on se restreint par convénience à des valeurs entières.

```
In [2]: type intervalle = (int * int);;
        type intervalles = intervalle list;;

Out[2]: type intervalle = int * int


Out[2]: type intervalles = intervalle list
```

- Pour des **graphes d'intervalles**, on utilise une simple représentation sous forme de liste d'adjacence, plus facile à mettre en place en OCaml qu'une représentation sous forme de matrice. Ici, tous nos graphes ont pour sommets $0 \dots n-1$.

```
In [3]: type sommet = int;;
        type voisins = sommet list;;
        type graphe_intervalle = voisins list;;

Out[3]: type sommet = int


Out[3]: type voisins = sommet list
```

```
Out[3]: type graphe_intervalle = voisins list
```

Note: j'ai préféré garder une structure très simple, pour les intervalles, les graphes d'intervalles et les coloriages, mais on perd un peu en lisibilité dans la fonction coloriage.

Implicitement, dès qu'une liste d'intervalles est fixée, de taille n, ils sont numérotés de 0 à $n-1$. Le graphe `g` aura pour sommet $0 \dots n-1$, et le coloriage sera un simple tableau de couleurs `c` (i.e., d'entiers), donnant en `c.(i)` la couleur de l'intervalle numéro `i`.

Une solution plus intelligente aurait été d'utiliser des tables d'association, cf. le module Map de OCaml, et le code proposé par Julien durant son oral.

- On peut rapidement écrire une fonction qui va convertir une liste d'intervalle (`intervalles`) en un graphe d'intervalle. On crée les sommets du graphes, via `index_intvls` qui associe un intervalle à son indice, et ensuite on ajoute les arêtes au graphe selon les contraintes définissant un graphe d'intervalle :

$$\forall I, I' \in V, (I, I') \in E \Leftrightarrow I \neq I' \text{ and } I \cap I' \neq \emptyset$$

Donc avec des intervales $I = [x, y]$ et $I' = [a, b]$, cela donne :

$$\forall I = [x, y], I' = [a, b] \in V, (I, I') \in E \Leftrightarrow (x, y) \neq (a, b) \text{ and } \neg(b < x \text{ or } y < a)$$

```
In [4]: let graphe_depuis_intervalles (intvls : intervalles) : graphe_intervalle =
            let n = List.length intvls in                    (* Nomber de sommet *)
            let array_intvls = Array.of_list intvls in       (* Tableau des intervalles *)
            let index_intvls = Array.to_list (
                Array.init n (fun i -> (
                    array_intvls.(i), i)                     (* Associe un intervalle à son ind
                )
            ) in
            let gr = List.map (fun (a, b) ->                 (* Pour chaque intervalle [a, b] *
                List.filter (fun (x, y) ->                   (* On ajoute [x, y] s'il intersect
                    (x, y) <> (a, b)                         (* Intervalle différent *)
                    && not ( (b < x) || (y < a) )            (* pas x---y a---b ni a---b x---y
                ) intvls
            ) intvls in
            (* On transforme la liste de liste d'intervalles en une liste de liste d'entiers *
            List.map (fun voisins ->
                List.map (fun sommet ->                      (* Grace au tableau index_intvls *
                    List.assoc sommet index_intvls
                ) voisins
            ) gr
        ;;
```

```
Out[4]: val graphe_depuis_intervalles : intervalles -> graphe_intervalle = <fun>
```

2.3.2 Algorithme de coloriage de graphe d'intervalles

Étant donné un graphe $G = (V, E)$, on cherche un entier n minimal et une fonction $c : V \to \{1, \cdots, n\}$ telle que si $(v_1, v_2) \in E$, alors $c(v_1) \neq c(v_2)$.

On suit les indications de l'énoncé pour implémenter facilement cet algorithme.

Une *heuristique* simple pour résoudre ce problème consiste à appliquer l'algorithme glouton suivant : - tant qu'il reste reste des sommets non coloriés, + en choisir un + et le colorier avec le plus petit entier qui n'apparait pas dans les voisins déjà coloriés.

En choisissant bien le nouveau sommet à colorier à chaque fois, cette heuristique se révelle optimale pour les graphes d'intervalles.

On peut d'abord définir un type de donnée pour un coloriage, sous la forme d'une liste de couple d'intervalle et de couleur. Ainsi, `List.assoc` peut être utilisée pour donner le coloriage de chaque intervalle.

```
In [5]: type couleur = int;;
        type coloriage = (intervalle * couleur) list;;

        let coloriage_depuis_couleurs (intvl : intervalles) (c : couleur array) : coloriage =
            Array.to_list (Array.init (Array.length c) (fun i -> (List.nth intvl i), c.(i)));;

        let quelle_couleur (intvl : intervalle) (colors : coloriage) =
            List.assoc intvl colors
        ;;
```

```
Out[5]: type couleur = int
```

```
Out[5]: type coloriage = (intervalle * couleur) list
```

```
Out[5]: val coloriage_depuis_couleurs : intervalles -> couleur array -> coloriage =
          <fun>
```

```
Out[5]: val quelle_couleur : intervalle -> coloriage -> couleur = <fun>
```

Ensuite, l'ordre partiel $\prec_i$ sur les intervalles est défini comme ça :

$$I = (a, b) \prec_i J = (x, y) \iff a < x$$

```
In [8]: let ordre_partiel ((a, _) : intervalle) ((x, _) : intervalle) =
            a < x
        ;;
```

```
Out[8]: val ordre_partiel : intervalle -> intervalle -> bool = <fun>
```

On a ensuite besoin d'une fonction qui va calculer l'inf de $\mathbb{N} \setminus \{x : x \in \text{valeurs}\}$:

```
In [9]: let inf_N_minus valeurs =
            let res = ref 0 in  (* Très important d'utiliser une référence ! *)
            while List.mem !res valeurs do
                incr res;
            done;
            !res
        ;;

Out[9]: val inf_N_minus : int list -> int = <fun>
```

On vérifie rapidement sur deux exemples :

```
In [10]: inf_N_minus [0; 1; 3];;  (* 2 *)
         inf_N_minus [0; 1; 2; 3; 4; 5; 6; 10];; (* 7 *)

Out[10]: - : int = 2


Out[10]: - : int = 7
```

Enfin, on a besoin d'une fonction pour trouver l'intervalle $I \in V$, minimal pour $\prec_i$, tel que $c(I) = +\infty$.

```
In [11]: let trouve_min_interval intvl (c : coloriage) (inf : couleur) =
             let colorie inter = quelle_couleur inter c in
             (* D'abord on extraie {I : c(I) = +oo} *)
             let intvl2 = List.filter (fun i -> (colorie i) = inf) intvl in
             (* Puis on parcourt la liste et on garde le plus petit pour l'ordre *)
             let i0 = ref 0 in
             for j = 1 to (List.length intvl2) - 1 do
                 if ordre_partiel (List.nth intvl2 j) (List.nth intvl2 !i0) then
                     i0 := j;
             done;
             List.nth intvl2 !i0;
         ;;

Out[11]: val trouve_min_interval :
           intervalle list -> coloriage -> couleur -> intervalle = <fun>
```

Et donc tout cela permet de finir l'algorithme, tel que décrit dans le texte :

```
In [12]: let coloriage_intervalles (intvl : intervalles) : coloriage =
             let n = List.length intvl in   (* Nombre d'intervalles *)
             let array_intvls = Array.of_list intvl in      (* Tableau des intervalles *)
             let index_intvls = Array.to_list (
```

```
        Array.init n (fun i -> (
            array_intvls.(i), i)                    (* Associe un intervalle à son in
        )
    ) in
    let gr = graphe_depuis_intervalles intvl in
    let inf = n + 10000 in           (* Grande valeur, pour +oo *)
    let c = Array.make n inf in      (* Liste des couleurs, c(I) = +oo pour tout I
    let maxarray = Array.fold_left max (-inf - 10000) in  (* Initialisé à -oo *)
    while maxarray c = inf do      (* Il reste un I in V tel que c(I) = +oo *)
        begin (* C'est la partie pas élégante *)
            (* On récupère le coloriage depuis la liste de couleurs actuelle *)
            let coloriage = (coloriage_depuis_couleurs intvl c) in
            (* Puis la fonction [colorie] pour associer une couleur à un intervalle *
            let colorie inter = quelle_couleur inter coloriage in
            (* On choisit un I, minimal pour ordre_partiel, tel que c(I) = +oo *)
            let inter = trouve_min_interval intvl coloriage inf in
            (* On trouve son indice *)
            let i = List.assoc inter index_intvls in
            (* On trouve les voisins de i dans le graphe *)
            let adj_de_i = List.nth gr i in
            (* Puis les voisins de I en tant qu'intervalles *)
            let adj_de_I = List.map (fun j -> List.nth intvl j) adj_de_i in
            (* Puis on récupère leurs couleurs *)
            let valeurs = List.map colorie adj_de_I in
            (* c(I) = inf(N - {c(J) : J adjacent a I} ) *)
            c.(i) <- inf_N_minus valeurs;
        end;
    done;
    coloriage_depuis_couleurs intvl c;
;;
```

```
Out[12]: val coloriage_intervalles : intervalles -> coloriage = <fun>
```

Une fois qu'on a un coloriage, à valeurs dans $0, \ldots, k$ on récupère le nombre de couleurs comme $1 + \max c$, i.e., $k + 1$.

```
In [13]: let max_valeurs = List.fold_left max 0;;
```

```
Out[13]: val max_valeurs : int list -> int = <fun>
```

```
In [14]: let nombre_chromatique (colorg : coloriage) : int =
             1 + max_valeurs (List.map snd colorg)
         ;;
```

```
Out[14]: val nombre_chromatique : coloriage -> int = <fun>
```

2.3.3 Algorithme pour calculer le *stable maximum* d'un graphe d'intervalles

On répond ici à la question 7.

> ń Proposer un algorithme efficace pour construire un stable maximum (i.e., un ensemble de sommets indépendants) d'un graphe d'intervalles dont on connat une représentation sous forme d'intervalles. On pourra chercher à quelle condition l'intervalle dont l'extrémité droite est la plus à gauche appartient à un stable maximum. ż

FIXME, je ne l'ai pas encore fait.

2.4 Exemples

On traite ici l'exemple introductif, ainsi que les trois autres exemples proposés.

2.4.1 Qui a tué le Duc de Densmore ?

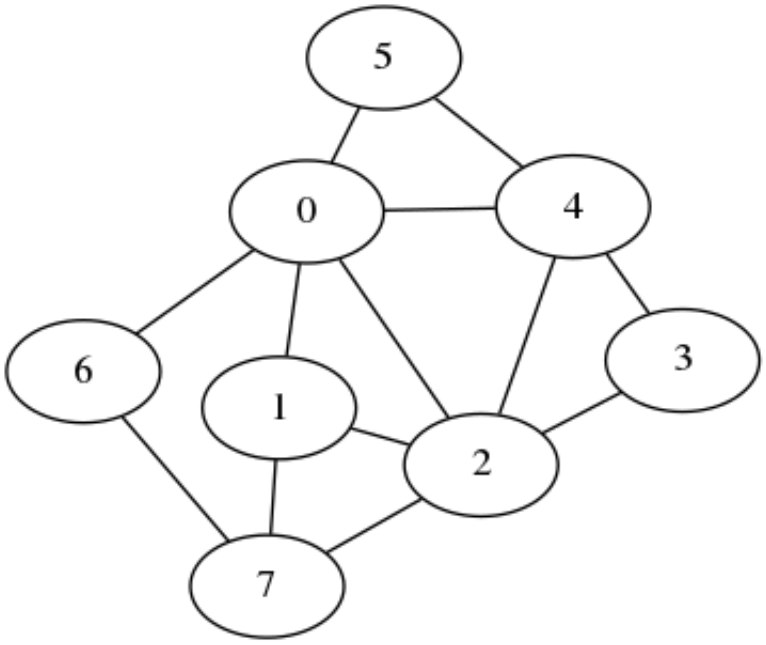
On ne rappelle pas le problème, mais les données :

- Ann dit avoir vu Betty, Cynthia, Emily, Felicia et Georgia.
- Betty dit avoir vu Ann, Cynthia et Helen.
- Cynthia dit avoir vu Ann, Betty, Diana, Emily et Helen.
- Diana dit avoir vu Cynthia et Emily.
- Emily dit avoir vu Ann, Cynthia, Diana et Felicia.
- Felicia dit avoir vu Ann et Emily.
- Georgia dit avoir vu Ann et Helen.
- Helen dit avoir vu Betty, Cynthia et Georgia.

Transcrit sous forme de graphe, cela donne :

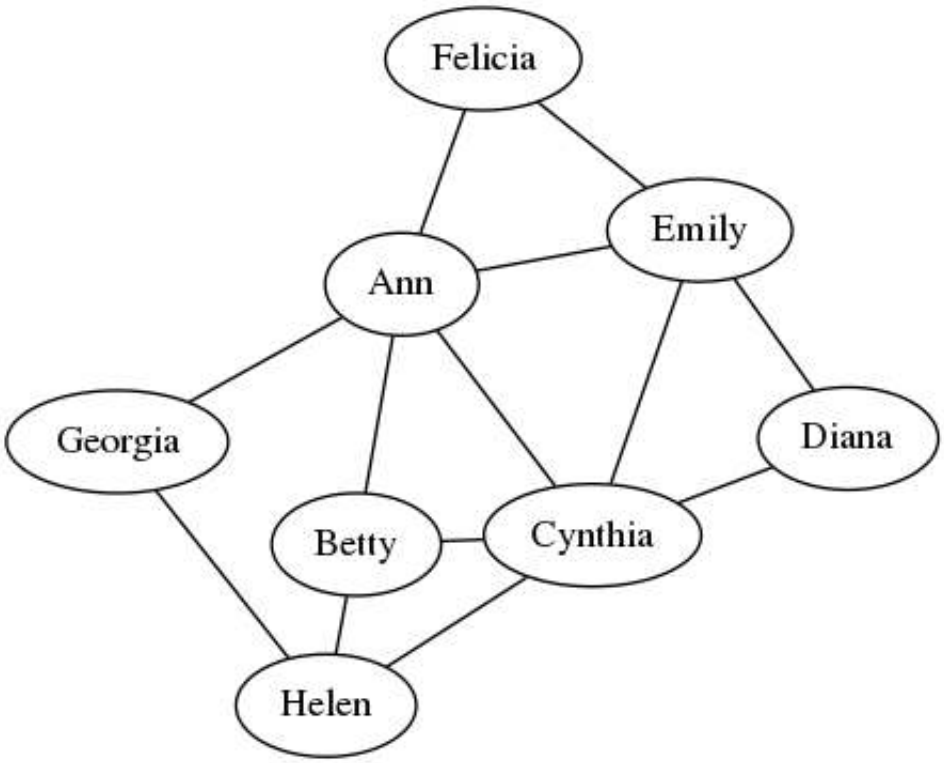
```
In [15]: (* On définit des entiers, c'est plus simple *)
         let ann = 0
         and betty = 1
         and cynthia = 2
         and diana = 3
         and emily = 4
         and felicia = 5
         and georgia = 6
         and helen = 7;;

         let graphe_densmore = [
             [betty; cynthia; emily; felicia; georgia];  (* Ann *)
             [ann; cynthia; helen]; (* Betty *)
             [ann; betty; diana; emily; helen]; (* Cynthia *)
             [cynthia; emily]; (* Diana *)
             [ann; cynthia; diana; felicia]; (* Emily *)
             [ann; emily]; (* Felicia *)
             [ann; helen]; (* Georgia *)
```

```
            [betty; cynthia; georgia] (* Helen *)
        ];;

Out[15]: val ann : int = 0
         val betty : int = 1
         val cynthia : int = 2
         val diana : int = 3
         val emily : int = 4
         val felicia : int = 5
         val georgia : int = 6
         val helen : int = 7


Out[15]: val graphe_densmore : int list list =
           [[1; 2; 4; 5; 6]; [0; 2; 7]; [0; 1; 3; 4; 7]; [2; 4]; [0; 2; 3; 5];
            [0; 4]; [0; 7]; [1; 2; 6]]
```



> Figure 1. Graphe d'intervalle pour le problème de l'assassinat du duc de Densmore.

Avec les prénoms plutôt que des numéros, cela donne :



> Figure 2. Graphe d'intervalle pour le problème de l'assassinat du duc de Densmore.

Comment résoudre ce problème ?

> Il faut utiliser la caractérisation du théorème 2 du texte, et la définition des graphes parfaits.

- Définition + Théorème 2 (point 1) :

On sait qu'un graphe d'intervalle est parfait, et donc tous ses graphes induits le sont aussi. La caractérisation via les cordes sur les cycles de taille ≥ 4 permet de dire qu'un quadrilatère (cycle de taille 4) n'est pas un graphe d'intervalle. Donc un graphe qui contient un graphe induit étant un quadrilatère ne peut être un graphe d'intervalle.

Ainsi, sur cet exemple, comme on a deux quadrilatères $ABHG$ et $AGHC$, on en déduit que A, G, ou H ont menti.

- Théorème 2 (point 2) :

Ensuite, si on enlève G ou H, le graphe ne devient pas un graphe d'intervalle, par les considérations suivantes, parce que son complémentaire n'est pas un graphe de comparaison.

En effet, par exemple si on enlève G, A et H et D forment une clique dans le complémentaire $\overline{G}$ de G, et l'irréflexivité d'une éventuelle relation R rend cela impossible. Pareil si on enlève H, avec G et B et D qui formet une clique dans $\overline{G}$.

Par contre, si on enlève A, le graphe devient triangulé (et de comparaison, mais c'est plus dur à voir !).

Donc seule A reste comme potentielle menteuse.

ń Mais... Ça semble difficile de programmer une résolution automatique de ce problème ? ż

En fait, il suffit d'écrire une fonction de vérification qu'un graphe est un graphe d'intervalle, puis on essaie d'enlever chaque sommet, tant que le graphe n'est pas un graphe d'intervalle.

Si le graphe devient valide en enlevant un seul sommet, et qu'il n'y en a qu'un seul qui fonctionne, alors il y a un(e) seul(e) menteur(se) dans le graphe, et donc un(e) seul(e) coupable !

Solution C'est donc A, i.e., Ann l'unique menteuse et donc la coupable.

Ce n'est pas grave de ne pas avoir réussi à répondre durant l'oral ! Au contraire, vous avez le droit de vous détacher du problème initial du texte !

Une solution bien expliquée peut être trouvée dans cette vidéo :

2.4.2 Le problème des frigos

Dans un grand hopital, les réductions de financement public poussent le gestionnaire du service d'immunologie à faire des économies sur le nombre de frigos à acheter pour stocker les vaccins. A peu de chose près, il lui faut stocker les vaccins suivants :

>	Numéro	Nom du vaccin
0	Rougeole-Rubéole-Oreillons (RRO)	$4 \cdots 12$ řC
1	BCG	$8 \cdots 15$ řC
2	Di-Te-Per	$0 \cdots 20$ řC
3	Anti-polio	$2 \cdots 3$ řC
4	Anti-hépatite B	$-3 \cdots 6$ řC
5	Anti-amarile	$-10 \cdots 10$ řC
6	Variole	$6 \cdots 20$ řC
7	Varicelle	$-5 \cdots 2$ řC
8	Antihaemophilus	$-2 \cdots 8$ řC

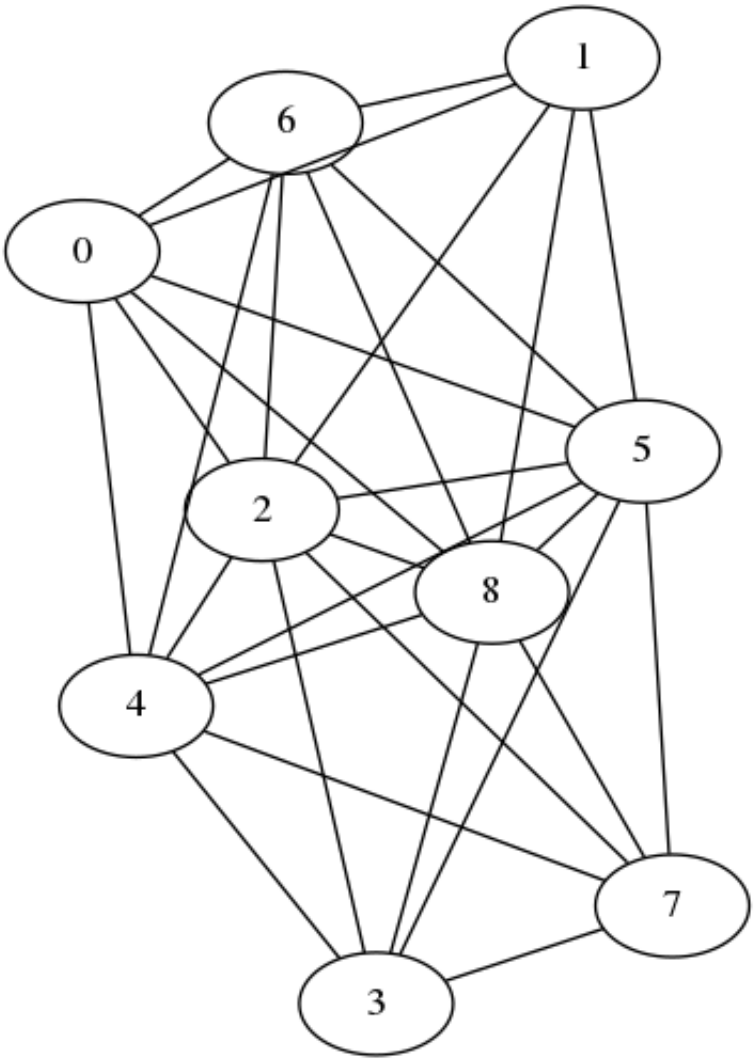
Combien le gestionaire doit-il acheter de frigos, et sur quelles températures doit-il les régler ?

```
In [16]: let vaccins : intervalles = [
             (4, 12);
             (8, 15);
             (0, 20);
             (2, 3);
             (-3, 6);
             (-10, 10);
             (6, 20);
             (-5, 2);
             (-2, 8)
         ]
```

```
Out[16]: val vaccins : intervalles =
           [(4, 12); (8, 15); (0, 20); (2, 3); (-3, 6); (-10, 10); (6, 20); (-5, 2);
            (-2, 8)]
```

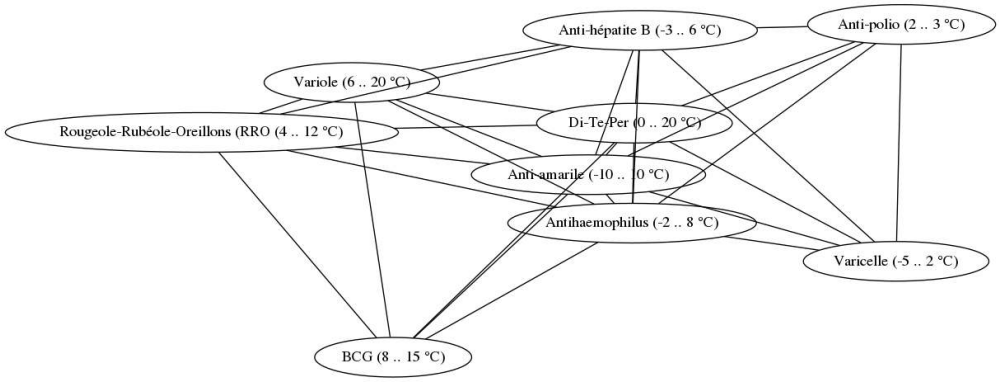
Qu'on peut visualiser sous forme de graphe facilement :

```
In [17]: let graphe_vaccins = graphe_depuis_intervalles vaccins;;

Out[17]: val graphe_vaccins : graphe_intervalle =
           [[1; 2; 4; 5; 6; 8]; [0; 2; 5; 6; 8]; [0; 1; 3; 4; 5; 6; 7; 8];
            [2; 4; 5; 7; 8]; [0; 2; 3; 5; 6; 7; 8]; [0; 1; 2; 3; 4; 6; 7; 8];
            [0; 1; 2; 4; 5; 8]; [2; 3; 4; 5; 8]; [0; 1; 2; 3; 4; 5; 6; 7]]
```



> Figure 3. Graphe d'intervalle pour le problème des frigos et des vaccins.

Avec des intervalles au lieu de numéro :



> Figure 4. Graphe d'intervalle pour le problème des frigos et des vaccins.

On peut récupérer une coloriage minimal pour ce graphe :

```
In [18]: coloriage_intervalles vaccins;;

Out[18]: - : coloriage =
         [((4, 12), 1); ((8, 15), 2); ((0, 20), 4); ((2, 3), 5); ((-3, 6), 2);
          ((-10, 10), 0); ((6, 20), 5); ((-5, 2), 1); ((-2, 8), 3)]
```

La couleur la plus grande est `5`, donc le nombre chromatique de ce graphe est `6`.

```
In [19]: nombre_chromatique (coloriage_intervalles vaccins);;

Out[19]: - : int = 6
```

Par contre, la solution au problème des frigos et des vaccins réside dans le nombre de couverture de cliques, $k(G)$, pas dans le nombre chromatique $\chi(G)$.

On peut le résoudre en répondant à la question 7, qui demandait de mettre au point un algorithme pour construire un *stable maximum* pour un graphe d'intervalle.

2.4.3 Le problème du CSA

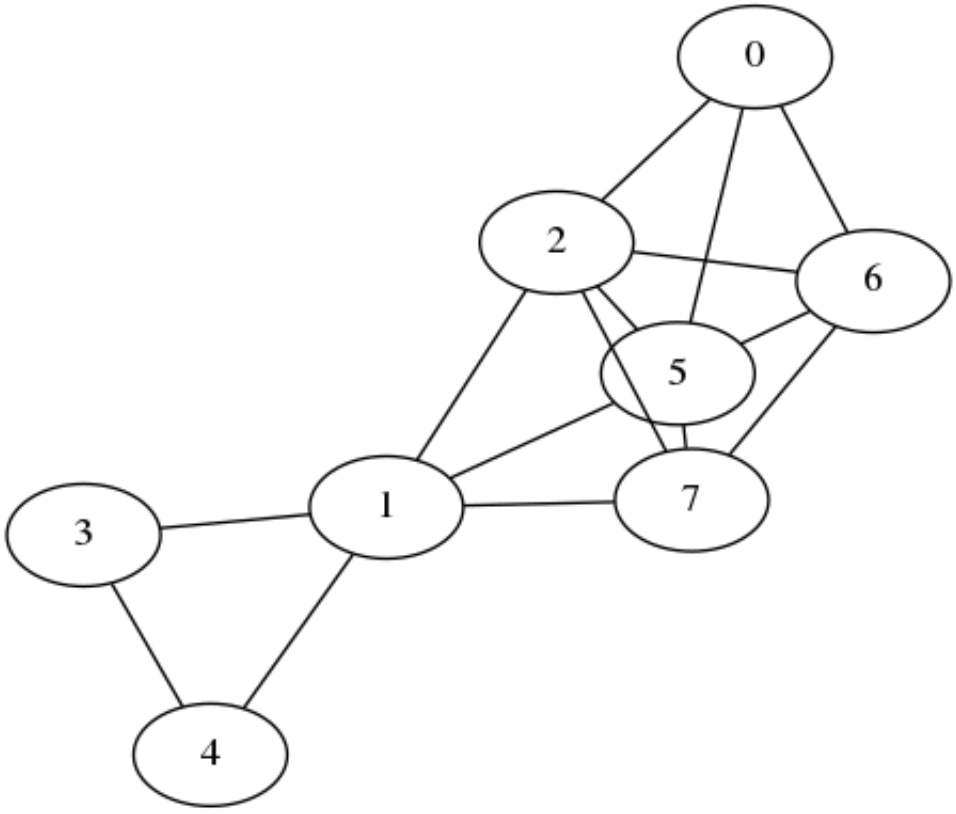
> Le Conseil Supérieur de l'Audiovisuel doit attribuer de nouvelles bandes de fréquences d'émission pour la stéréophonie numérique sous-terraine (SNS). Cette technologie de pointe étant encore à l'état expérimental, les appareils capables d'émettre ne peuvent utiliser que les bandes de fréquences FM suivantes :

>	Bandes de fréquence
0	$32 \cdots 36$
1	$24 \cdots 30$
2	$28 \cdots 33$
3	$22 \cdots 26$
4	$20 \cdots 25$
5	$30 \cdots 33$
6	$31 \cdots 34$
7	$27 \cdots 31$

Quelles bandes de fréquences doit-on retenir pour permettre à le plus d'appareils possibles d'être utilisés, sachant que deux appareils dont les bandes de fréquences s'intersectent pleinement (pas juste sur les extrémités) sont incompatibles.

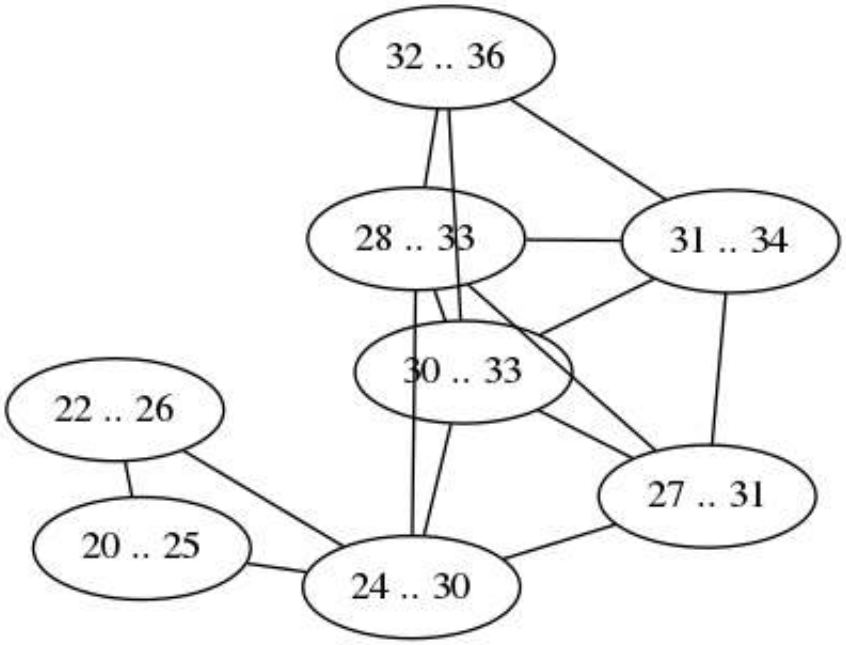
```
In [20]: let csa : intervalles = [
             (32, 36);
             (24, 30);
             (28, 33);
             (22, 26);
             (20, 25);
             (30, 33);
             (31, 34);
             (27, 31)
         ];;

Out[20]: val csa : intervalles =
           [(32, 36); (24, 30); (28, 33); (22, 26); (20, 25); (30, 33); (31, 34);
            (27, 31)]
```

```
In [21]: let graphe_csa = graphe_depuis_intervalles csa;;

Out[21]: val graphe_csa : graphe_intervalle =
           [[2; 5; 6]; [2; 3; 4; 5; 7]; [0; 1; 5; 6; 7]; [1; 4]; [1; 3];
            [0; 1; 2; 6; 7]; [0; 2; 5; 7]; [1; 2; 5; 6]]
```



> Figure 5. Graphe d'intervalle pour le problème du CSA.

Avec des intervalles au lieu de numéro :



> Figure 6. Graphe d'intervalle pour le problème du CSA.

On peut récupérer une coloriage minimal pour ce graphe :

```
In [22]: coloriage_intervalles csa;;

Out[22]: - : coloriage =
         [((32, 36), 0); ((24, 30), 2); ((28, 33), 1); ((22, 26), 1); ((20, 25), 0);
          ((30, 33), 3); ((31, 34), 2); ((27, 31), 0)]
```

La couleur la plus grande est 3, donc le nombre chromatique de ce graphe est 4.

```
In [23]: nombre_chromatique (coloriage_intervalles csa);;

Out[23]: - : int = 4
```

Par contre, la solution au problème CSA réside dans le nombre de couverture de cliques, $k(G)$, pas dans le nombre chromatique $\chi(G)$.

On peut le résoudre en répondant à la question 7, qui demandait de mettre au point un algorithme pour construire un *stable maximum* pour un graphe d'intervalle.

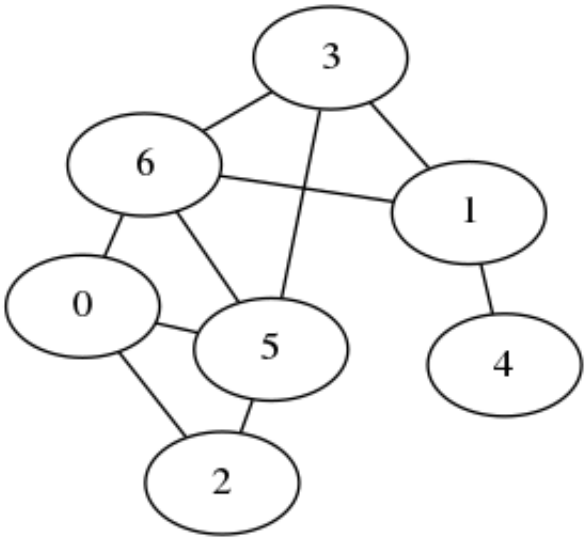
2.4.4 Le problème du wagon restaurant

> Le chef de train de l'Orient Express doit aménager le wagon restaurant avant le départ du train. Ce wagon est assez petit et doit être le moins encombré de tables possibles, mais il faut prévoir suffisemment de tables pour accueillir toutes personnes qui ont réservé :

>	Numéro	Personnage(s)	Heures de dîner
0	Le baron et la baronne Von Haussplatz	19h30 .. 20h14	1170 · · · 1214
1	Le général Cook	20h30 .. 21h59	1230 · · · 1319
2	Les époux Steinberg	19h .. 19h59	1140 · · · 1199
3	La duchesse de Colombart	20h15 .. 20h59	1215 · · · 1259
4	Le marquis de Carquamba	21h .. 21h59	1260 · · · 1319
5	La Vociafiore	19h15 .. 20h29	1155 · · · 1229
6	Le colonel Ferdinand	20h .. 20h59	1200 · · · 1259

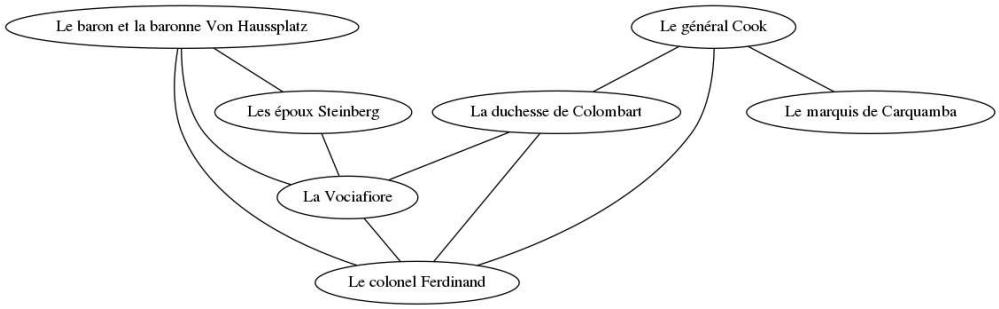
Combien de tables le chef de train doit-il prévoir ?

```
In [24]: let restaurant = [
             (1170, 1214);
             (1230, 1319);
             (1140, 1199);
             (1215, 1259);
             (1260, 1319);
             (1155, 1229);
             (1200, 1259)
         ];;

Out[24]: val restaurant : (int * int) list =
           [(1170, 1214); (1230, 1319); (1140, 1199); (1215, 1259); (1260, 1319);
            (1155, 1229); (1200, 1259)]


In [25]: let graphe_restaurant = graphe_depuis_intervalles restaurant;;

Out[25]: val graphe_restaurant : graphe_intervalle =
           [[2; 5; 6]; [3; 4; 6]; [0; 5]; [1; 5; 6]; [1]; [0; 2; 3; 6]; [0; 1; 3; 5]]
```



> Figure 7. Graphe d'intervalle pour le problème du wagon restaurant.

Avec des intervalles au lieu de numéro :



> Figure 8. Graphe d'intervalle pour le problème du wagon restaurant.

```
In [26]: coloriage_intervalles restaurant;;

Out[26]: - : coloriage =
         [((1170, 1214), 2); ((1230, 1319), 1); ((1140, 1199), 0); ((1215, 1259), 2);
          ((1260, 1319), 0); ((1155, 1229), 1); ((1200, 1259), 0)]
```

La couleur la plus grande est 2, donc le nombre chromatique de ce graphe est 3.

```
In [27]: nombre_chromatique (coloriage_intervalles restaurant);;

Out[27]: - : int = 3
```

Solution via l'algorithme de coloriage de graphe d'intervalles Pour ce problème là, la solution est effectivement donnée par le nombre chromatique.

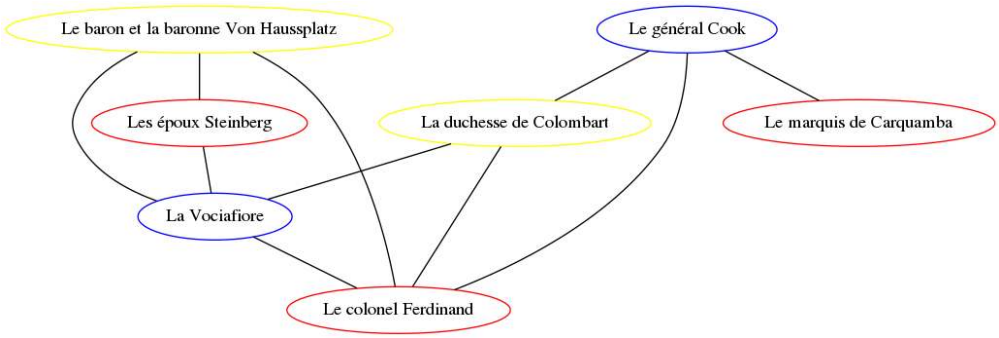
La couleur sera le numéro de table pour chaque passagers (ou couple de passagers), et donc le nombre minimal de table à installer dans le wagon restaurant est exactement le nombre chromatique.

Une solution peut être la suivante, avec **3 tables** :

Numéro	Personnage(s)	Heures de dîner	Numéro de table
0	Le baron et la baronne Von Haussplatz	19h30 .. 20h14	2
1	Le général Cook	20h30 .. 21h59	1
2	Les époux Steinberg	19h .. 19h59	0
3	La duchesse de Colombart	20h15 .. 20h59	2
4	Le marquis de Carquamba	21h .. 21h59	0
5	La Vociafiore	19h15 .. 20h29	1
6	Le colonel Ferdinand	20h .. 20h59	0

On vérifie manuellement que la solution convient. Chaque passager devra quitter sa tableau à la minute près par contre !

On peut afficher la solution avec un graphe colorié. La table 0 sera rouge, 1 sera bleu et 2 sera jaune :



> Figure 9. Solution pour le problème du wagon restaurant.

2.5 Bonus ?

2.5.1 Visualisation des graphes définis dans les exemples

- J'utilise une petite fonction facile à écrire, qui convertit un graphe (`int list list`) en une chaîne de caractère au format DOT Graph.
- Ensuite, un appel `dot -Tpng ...` en ligne de commande convertit ce graphe en une image, que j'inclus ensuite manuellement.

```
In [30]: (** Transforme un [graph] en une chaîne représentant un graphe décrit par le langage
             voir http://en.wikipedia.org/wiki/DOT_language pour plus de détails sur ce langag

             @param graphname Donne le nom du graphe tel que précisé pour DOT
             @param directed Vrai si le graphe doit être dirigé (c'est le cas ici) faux sinon
             @param verb Affiche tout dans le terminal.
             @param onetoone Si on veut afficher le graphe en mode carré (échelle 1:1). Parfo
         *)
         let graph_to_dotgraph ?(graphname = "graphname") ?(directed = false) ?(verb = false)
             let res = ref "" in
             let log s =
                 if verb then print_string s; (* Si [verb] affiche dans le terminal le résulta
                 res := !res ^ s
             in
             log (if directed then "digraph " else "graph ");
             log graphname; log " {";
             if onetoone then
                 log "\n   size=\"1,1\";";
             let g = Array.of_list (List.map Array.of_list glist) in
             (* On affiche directement les arc, un à un. *)
             for i = 0 to (Array.length g) - 1 do
                 for j = 0 to (Array.length g.(i)) - 1 do
                     if i < g.(i).(j) then
                         log ("\n   \""
                         ^ (string_of_int i) ^ "\" "
                         ^ (if directed then "->" else "--")
```

```
                    ^ " \"" ^ (string_of_int g.(i).(j)) ^ "\""
                    );
            done;
        done;
        log "\n}\n// generated by OCaml with the function graphe_to_dotgraph.";
    !res;;
```

```
Out[30]: val graph_to_dotgraph :
           ?graphname:string ->
           ?directed:bool -> ?verb:bool -> ?onetoone:bool -> int list list -> string =
           <fun>
```

```
In [29]: (** Fonction ecrire_sortie : plus pratique que output. *)
         let ecrire_sortie monoutchanel machaine =
             output monoutchanel machaine 0 (String.length machaine);
             flush monoutchanel;;

         (** Fonction ecrire_dans_fichier : pour écrire la chaine dans le fichier à l'adresse
         let ecrire_dans_fichier ~chaine ~adresse =
             let mon_out_channel = open_out adresse in
             ecrire_sortie mon_out_channel chaine;
             close_out mon_out_channel;;
```

```
Out[29]: val ecrire_sortie : out_channel -> bytes -> unit = <fun>
```

```
Out[29]: val ecrire_dans_fichier : chaine:bytes -> adresse:string -> unit = <fun>
```

```
In [31]: let s_graphe_densmore = graph_to_dotgraph ~graphname:"densmore" ~directed:false ~verb
         let s_graphe_vaccins = graph_to_dotgraph ~graphname:"vaccins" ~directed:false ~verb:fa
         let s_graphe_csa = graph_to_dotgraph ~graphname:"csa" ~directed:false ~verb:false grap
         let s_graphe_restaurant = graph_to_dotgraph ~graphname:"restaurant" ~directed:false ~v
```

```
Out[31]: val s_graphe_densmore : string =
           "graph densmore {\n   \"0\" -- \"1\"\n   \"0\" -- \"2\"\n   \"0\" -- \"4\"\n   \"0\
```

```
Out[31]: val s_graphe_vaccins : string =
           "graph vaccins {\n   \"0\" -- \"1\"\n   \"0\" -- \"2\"\n   \"0\" -- \"4\"\n   \"0\"
```

```
Out[31]: val s_graphe_csa : string =
           "graph csa {\n   \"0\" -- \"2\"\n   \"0\" -- \"5\"\n   \"0\" -- \"6\"\n   \"1\" -- \
```

```
Out[31]: val s_graphe_restaurant : string =
           "graph restaurant {\n   \"0\" -- \"2\"\n   \"0\" -- \"5\"\n   \"0\" -- \"6\"\n   \"
```

```
In [32]: ecrire_dans_fichier ~chaine:s_graphe_densmore ~adresse:"/tmp/densmore.dot" ;;
         (* Sys.command "fdp -Tpng /tmp/densmore.dot > images/densmore.png";; *)

         ecrire_dans_fichier ~chaine:s_graphe_vaccins ~adresse:"/tmp/vaccins.dot" ;;
         (* Sys.command "fdp -Tpng /tmp/vaccins.dot > images/vaccins.png";; *)

         ecrire_dans_fichier ~chaine:s_graphe_csa ~adresse:"/tmp/csa.dot" ;;
         (* Sys.command "fdp -Tpng /tmp/csa.dot > images/csa.png";; *)

         ecrire_dans_fichier ~chaine:s_graphe_restaurant ~adresse:"/tmp/restaurant.dot" ;;
         (* Sys.command "fdp -Tpng /tmp/restaurant.dot > images/restaurant.png";; *)

Out[32]: - : unit = ()


Out[32]: - : unit = ()


Out[32]: - : unit = ()


Out[32]: - : unit = ()
```

On pourrait étendre cette fonction pour qu'elle prenne les intervalles initiaux, pour afficher des bonnes étiquettes et pas des entiers, et un coloriage pour colorer directement les noeuds, mais ça prend du temps pour pas grand chose.

2.6 Conclusion

Voilà pour la question obligatoire de programmation, sur l'algorithme de coloriage.

- 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 de petits exemples et sur un exemple de taille réelle (venant du texte)

Et on a pas essayé de faire *un peu plus*. Avec plus de temps, on aurait aussi pu écrire un algorithme pour calculer le stable maximum (ensemble de sommets indépendants de taille maximale).

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