DS 2

Option informatique, deuxième année

Julien Reichert

On impose de donner le type, ou la signature, de chaque fonction écrite, sauf lorsqu'il ou elle est indiqué(e) par le sujet, auquel cas la réponse doit être d'un type compatible avec la signature proposée.

1 Graphes d'intervalles

On considère le problème concret suivant : des cours doivent avoir lieu dans un intervalle de temps précis (de 8h à 9h55,...) et on cherche à attribuer une salle à chaque cours. On souhaite qu'à tout moment une salle ne puisse être attribuée à deux cours différents et on aimerait utiliser le plus petit nombre de salles possibles. Ce problème d'allocation de ressources (ici les salles) en fonction de besoins fixes (ici les horaires des cours) intervient dans de nombreuses situations très diverses (allocation de pistes d'atterrissage aux avions, répartition de la charge de travail sur plusieurs machines,...).

1.1 Représentation du problème

On modélise le problème ainsi :
- chaque besoin est représenté par un segment $[a, b]$ où $a, b \in \mathbb{N}$ et $a \leq b$;
- deux besoins I et J sont en conflit quand $I \cap J \neq \emptyset$.

La donnée du problème est une suite finie $(I_0, \ldots, I_{n-1})$ de n segments où $n \in \mathbb{N} \setminus \{0\}$.

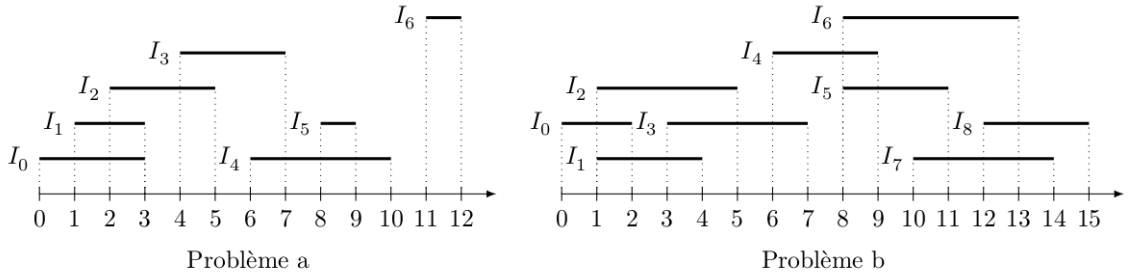


Figure 1 Deux exemples de problèmes

On représente un segment en Caml par un couple d'entiers, la donnée du problème est une valeur du type `(int*int) vect`. Le problème a de la figure 1 est représenté par le tableau

```
[| (0,3); (1,3); (2,5); (4,7); (6,10); (8,9); (11,12) |].
```

Question 1 : Écrire une fonction ayant pour signature `conflit : int * int -> int * int -> bool` telle que `conflit I J` renvoie `true` si et seulement si `I` et `J` sont en conflit.

1.2 Graphe simple non orienté

On appelle graphe simple non orienté un couple $G = (S, A)$ où :
- S est un ensemble fini dont les éléments sont appelés les sommets du graphe ;
- A est un ensemble de paires d'éléments distincts de S. Lorsque $\{x, y\} \in A$ on dit que x et y sont reliés dans G et $\{x, y\}$ est appelée une arête de G. Les sommets reliés à un sommet x sont appelés les voisins de x.

Étant donnée une énumération de S sous la forme d'une suite finie $(x_0, \ldots, x_{n-1})$, on représente A en Caml par un élément du type `int list vect` ainsi : pour `i` $\in \{0, \ldots, n-1\}$, la liste `A.(i)` contient les `j` tels que x_i soit relié à x_j dans G. On représente graphiquement le graphe G par un diagramme où les arêtes sont représentées par des traits entre les sommets.

Les arêtes du graphe dont une représentation graphique est donnée figure 2 sont représentées en Caml par le tableau :

```
[| [1;2;3]; [0;2;3]; [0;1;3;4]; [0;1;2]; [2] |]
```

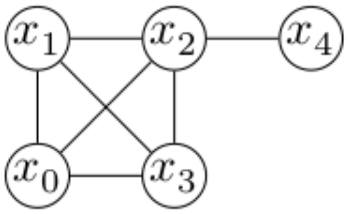


Figure 2

Une telle liste d'arêtes suffit pour déterminer un graphe lorsque l'énumération des sommets est connue car on peut alors identifier un sommet à son indice. Dans la suite de ce problème on identifiera ainsi un graphe à sa liste d'arêtes.

1.3 Graphe d'intervalles

Soit $\bar{I} = (I_0, \ldots, I_{n-1})$ une suite finie de segments, on appelle graphe d'intervalles associé à $\bar{I}$ le graphe $G(\bar{I})$
- dont les sommets sont les segments I_0 , …, I_{n-1} ;
- et où, pour $i, j \in \{0, \ldots, n-1\}$, avec $i \neq j$, les sommets I_i et I_j sont reliés si et seulement si ils sont en conflit.

Le graphe d'intervalles qui correspond au problème a de la figure 1 admet la représentation graphique de la figure 3.

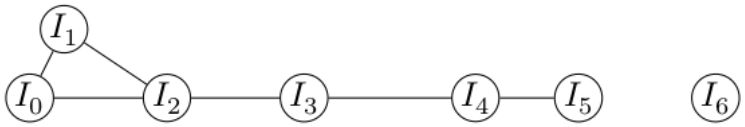


Figure 3

Question 2 : Donner une représentation graphique du graphe d'intervalles associé au problème b de la figure 1.

Question 3 : Écrire une fonction ayant pour signature `construit_graphe : (int * int) vect -> int list vect` qui étant donné le tableau des segments $\bar{I} = (I_0, \ldots, I_{n-1})$, énumérés dans cet ordre, renvoie la représentation des arêtes de $G(\bar{I})$.

1.4 Coloration

Soit $G = (S, A)$ un graphe simple non orienté dont les sommets sont $x_0, \ldots, x_{n-1}$. On appelle coloration de G une suite finie d'entiers naturels $(c_0, \ldots, c_{n-1})$ telle que

$$\forall i, j \in \{0, \ldots, n-1\}, \{x_i, x_j\} \in A \Rightarrow c_i \neq c_j.$$

L'entier c_i est appelé la couleur du sommet x_i et la condition se traduit ainsi : deux sommets reliés ont des couleurs distinctes. Dorénavant, le terme couleur sera synonyme d'entier naturel.

La suite finie $(0, 1, 2, 3, 0)$ est une coloration du graphe de la figure 2.

Lorsqu'une coloration utilise le plus petit nombre de couleurs distinctes possibles, on dit qu'elle est optimale. On note alors $\chi(G)$ ce nombre minimum de couleurs, appelé le nombre chromatique de G.

En associant une salle à chaque couleur, on peut répondre au problème initial à l'aide d'une coloration de son graphe d'intervalles associé.

Question 4 : Déterminer des colorations optimales pour les graphes d'intervalles associés aux deux problèmes de la figure 1. On attribuera à chaque fois la couleur 0 à l'intervalle I_0.

Question 5 : Écrire deux fonctions, l'une de signature `appartient : int list -> int -> bool`, telle que l'appel à `appartient l x` envoie `true` si et seulement si l'entier `x` est présent dans la liste `l`, et l'autre de signature `plus_petit_absent : int list -> int`, telle que l'appel à `plus_petit_absent l` renvoie le plus petit entier naturel non présent dans `l`.

Question 6 : On considère ici une coloration progressive des sommets d'un graphe. Pour cela, une coloration partielle est un tableau `couleurs : int vect` tel que `couleurs.(i)` contient la couleur de `i` s'il est coloré et `-1` sinon, ce qui ne pose pas de problème car les couleurs sont toujours positives.
Écrire une fonction de signature `couleurs_voisins : int list vect -> int vect -> int -> int list` telle que l'appel à `couleurs_voisins aretes couleurs i` renvoie la liste des couleurs des voisins colorés du sommet d'indice `i` dans le graphe décrit par `aretes` où le tableau couleurs décrit une coloration partielle.

Question 7 : En déduire une fonction de signature

```
couleur_disponible : int list vect -> int vect -> int -> int
```

telle que l'appel à `couleur_disponible aretes couleurs i` renvoie la plus petite couleur pouvant être attribuée au sommet `i` afin qu'il n'ait la couleur d'aucun de ses voisins dans le graphe décrit par `aretes`.

1.5 Cliques

Soit $G = (S, A)$ un graphe. Un sous-ensemble $C \subseteq S$ est appelé une clique de G lorsqu'il vérifie

$$\forall x, y \in C, x \neq y \Rightarrow \{x, y\} \in A.$$

Le nombre d'éléments de C est appelé sa taille. La taille de la plus grande (celle qui possède le plus grand nombre d'éléments) clique de G est notée $\omega(G)$.

Question 8 : Déterminer $\chi(G)$ et $\omega(G)$ lorsque G ne possède pas d'arête (c'est à dire $A = \emptyset$), et lorsque G est un graphe complet à n sommets, c'est à dire $|S| = n$ et pour tous $u, v \in S$ distincts, $\{u, v\} \in A$.

Question 9 : Comparer $\chi(G)$ et $\omega(G)$ pour un graphe G quelconque.

Question 10 : Écrire une fonction de signature `est_clique : int list vect -> int list -> bool` telle que `est_clique aretes xs` renvoie `true` si et seulement si la liste `xs` est une liste d'indices de sommets formant une clique dans le graphe décrit par `aretes`.

1.6 Algorithme glouton pour la coloration

Étant donnée une liste de segments $\bar{I} = (I_0, I_1, \ldots, I_{n-1})$ de longueur $n \geq 1$, on se propose de déterminer une coloration optimale de son graphe d'intervalles associé. On appelle coloration de $\bar{I}$ une suite finie d'entiers naturels $(c_0, \ldots, c_{n-1})$ telle que

$$\forall i, j \in \{0, \ldots, n-1\}, I_i \cap I_j \neq \emptyset \Rightarrow c_i \neq c_j.$$

On suppose dans cette partie que les segments $I_k = [a_k, b_k]$, pour $k \in \{0, \ldots, n-1\}$, sont énumérés dans l'ordre croissant de leur extrémités gauches, c'est-à-dire que $a_0 \leq a_1 \leq \cdots \leq a_{n-1}$.

On propose l'algorithme suivant :

Pour k variant de 0 à $n-1$, colorer l'intervalle I_k avec la plus petite couleur non encore utilisée dans la coloration des intervalles I_j, avec $0 \leq j < k$, qui ont une intersection non vide avec I_k.

Ainsi, l'intervalle I_0 est toujours coloré avec la couleur 0, l'intervalle I_1 reçoit la couleur 0 si $I_0 \cap I_1 = \emptyset$, et la couleur 1 sinon, etc.

Question 11 : Déterminer la coloration renvoyée par l'algorithme pour le problème b décrit sur la figure 1. Si par hasard elle a déjà été donnée à la question 4, on peut se contenter de le signaler tout simplement.

Question 12 : Écrire une fonction de signature `coloration : (int * int) vect -> int list vect -> int vect` telle que l'appel `coloration segments aretes`, où `segments` est un tableau contenant des segments triés par ordre croissant de leurs extrémités gauches et où `aretes` représente les arêtes du graphe d'intervalles associé à ces segments, renvoie la coloration obtenue avec l'algorithme ci-dessus.

On se propose maintenant de démontrer que l'algorithme ci-dessus fournit une coloration optimale de l'ensemble de segments. Soit k un entier entre 0 et $n-1$. On suppose qu'à la k-ième étape de l'algorithme, le segment I_k reçoit la couleur c.

Question 13 : L'extrémité gauche du segment I_k appartient à un certain nombre de segments parmi $I_0, I_1, \ldots, I_{k-1}$. Combien au moins ?

Question 14 : Prouver que l'ensemble constitué de I_k et de ses voisins d'indice inférieur à k constitue une clique de taille au moins $c+1$ dans le graphe d'intervalles associé.

Question 15 : En déduire que le nombre de couleurs nécessaires à une coloration de l'ensemble des segments est au moins égal à $c+1$, puis conclure.

Question 16 : Déterminer la complexité de la fonction `coloration` en fonction du nombre m d'arêtes du graphe d'intervalles associé à la liste $\bar{I}$.

2 Graphe du Web

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) 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.

2.1 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.

Question 17 : 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` et de la forme $[y_{i1};\ldots;y_{ik_i}]$, `aplatir liste` est une liste d'éléments de type `'a` :

$$[x_1;y_{11};\ldots;y_{1k_1};\ldots;x_n;y_{n1};\ldots;y_{nk_n}].$$

Question 18 : 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. À 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.

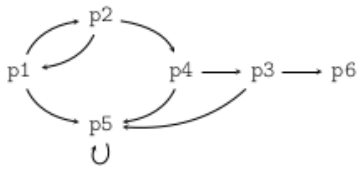
On suppose que le type de données `dictionnaire` est prédéfini ; on ne demande pas de l'implémenter.

Question 19 : 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`.

Question 20 : 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.

2.2 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, `p1`, `p2`, etc., sont les URL de pages Web (simplifiées pour l'exemple), et les arcs représentent les hyperliens. Dans ce mini-graphe, un appel à `recupere_liens "p1"` retourne `["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). À 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.

Question 21 : 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"]]
```

Question 22 : Même question mais avec une fonction `crawler_dfs` qui doit suivre 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.

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"]]
```

Question 23 : Coder une fonction Caml `construit_graphe : (string * string list) list -> string list * int vect vect` telle que si `crawl` est le résultat renvoyé par un crawler, alors `construit_graphe crawl` est un couple `(l, G)` où `l` est une liste de toutes les URL de pages constenues 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"`, 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, `p3` apparaît même s'il n'a pas été visité dans le crawl, mais `p6` n'apparaît pas car il n'a pas été découvert dans le crawl. En outre, l'hyperlien de `p3` à `p5` n'apparaît pas car `p3` n'a pas été visité.