L'énoncé contient un exercice et un problème qui sont indépendants. Il est conseillé de passer environ une heure sur l'exercice et trois heures sur le problème

En plus des fonctionnalités de base du langage OCaml, le candidat pourra utiliser les fonctions suivantes sans les programmer.

- `Array.make : int → 'a → 'a array` telle que `Array.make n v` renvoie un tableau de longueur n dont toutes les cases valent v.
- `Array.make_matrix : int → int → 'a → 'a array` telle que `Array.make n p v` renvoie une matrice ayant n lignes et p colonnes dont toutes les cases valent v.
- `Array.length : 'a array → int` elle que `Array.length t` renvoie la longueur de `t`.

Exercice

Le but de cet exercice est d'étudier l'algorithme de Bellman-Ford. Il permet de calculer le poids d'un chemin de plus faible poids à partir d'une source s'il n'y a pas de cycle de poids négatif dans le graphe et ce, même s'il existe des arêtes de poids négatif.

Soit n un entier non nul, on considère un graphe orienté $G = (S, A)$ où $S = \{0, 1, \dots n-1\}$. Le graphe est pondéré à l'aide d'une fonction $w : S \to \mathbb{R}$. On note $p = |A|$ le nombre d'arêtes du graphe. On supposera que la source sera le sommet $s = 0$.

Le but de l'algorithme est de construire une matrice $D = (d(i,k))_{(i,k)\in[\![0,n-1]\!]^2}$ telle que $d(i,k)$ contient (à la fin de l'exécution) le poids d'un chemin de poids minimal reliant le sommet source au sommet i et ayant au plus k arêtes.

L'algorithme est le suivant :

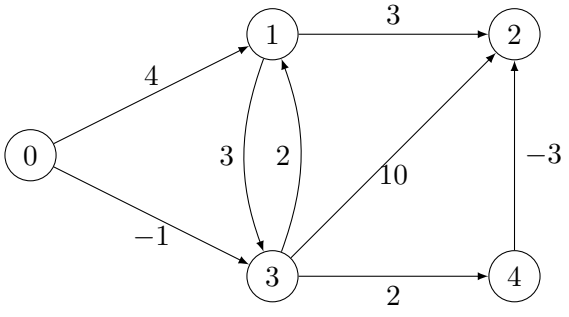
Algorithme 1 : Algorithme de Bellman-Ford

Données : Graphe $G = (S, A)$; sommet de départ $s = 0$

```
1  pour  i ∈ S faire
2  |  d[i, 0] ← ∞
3  fin
4  d[0, 0] ← 0
5  pour k ∈ [[1, |S| − 1]] faire
6  |  pour i ∈ S faire
7  |  |  d' ← d[i, k − 1]
8  |  |  pour (u, i) ∈ A faire
9  |  |  |  d' ← min(d', d[u, k − 1] + w(u, i))
10 |  |  fin
11 |  |  d[i, k] ← d'
12 |  fin
13 fin
```

Résultat : La colonne $d[., |S| - 1]$

Pour les exemples nous considérerons le graphe



1) Donner la matrice $D = (d(i,k))_{(i,k)\in[\![0,4]\!]^2}$ obtenue à la fin de l'exécution de l'algorithme sur le graphe de l'exemple. On justifiera les valeurs de $d(1,2)$ et $d(2,3)$.

2) Dans le cas général, expliquer (rapidement) pourquoi, à la fin de l'exécution de l'algorithme, $d(i,k)$ contient le poids d'un chemin de poids minimal reliant le sommet source au sommet i et ayant au plus k arêtes. Expliquer alors pourquoi le résultat renvoyé par l'algorithme est bien celui voulu.

   On demande juste une explication informelle.

3) Donner, à l'aide de n et p la complexité temporelle de l'algorithme.

Afin de pouvoir définir des distances infinies pour l'initialisation de l'algorithme, on utilise un type `int_Inf` défini par

```
type int_Inf = Infini | Entier of int
```

On veut implémenter cet algorithme. On suppose que le graphe G est donnée par des listes d'adjacences pondérées. C'est-à-dire un tableau `g` de type `(int*int list) array` tel que pour tout sommet `i`, la liste `g.(i)` soit la liste des couples (u,w) où l'arête (i,u) a un poids w. Par exemple, le graphe donné ci-dessus sera codé par

```
g = [|[(1,4);(3,-1)];[(2,3);(3,3)];[];[(1,2);(2,10);(4,2)];[(2,-3)]|].
```

4) Définir les fonctions

   ```
   add_Inf : int_Inf -> int_Inf -> int_Inf
   ```

   et

   ```
   min_Inf : int_Inf -> int_Inf -> int_Inf
   ```

   qui permettent respectivement d'ajouter deux éléments de type `int_Inf` ou de trouver le minimum de deux éléments de type `int_Inf`.

5) Pour implémenter notre algorithme, il sera plus efficace d'utiliser un tableau de listes de prédécesseurs plutôt que de listes de successeurs. Le tableau des listes des prédécesseurs d'un graphe G est un tableau `g` de type `(int*int list) array` tel que pour tout sommet `i`, la liste `g.(i)` soit la liste des couples (u,w) où l'arête (u,i) a un poids w dans le graphe.

   a) Donner le tableau des listes des prédécesseurs du graphe de l'exemple.

   b) Écrire une fonction `succ_prec : (int,int) list array -> (int,int) list array` tel que si `g` est le tableau des listes des successeurs d'un graphe, `succ_prec g` renvoie le tableau des listes des prédécesseurs. On précisera la complexité de la fonction. Pour avoir tous les points, la fonction proposées devra avoir une complexité linéaire en le nombre d'arêtes du graphe.

6) Écrire une fonction `bf : (int*int) list array -> int_Inf array` telle que si `g` est un un graphe donné par listes d'adjacence, `bf g` renvoie un tableau `t` de taille n (où n est la taille du graphe) tel que `t.(i)` contient le poids d'un chemin de poids minimal dans G allant du sommet 0 au sommet i.

Problème

Notations et conventions

Par **complexité en temps** d'un algorithme A, on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc) nécessaires à l'exécution de A dans le cas le pire.

Lorsque la complexité en temps ou en espace dépend d'un n, on dit que A a une complexité $O(f(n))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de n suffisamment grandes (c'est à dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètre n, la complexité est au plus $Cf(n)$.

On dit que la complexité en temps est **linéaire** quand f est une fonction linéaire du paramètre n, **polynomiale** quand f est une fonction polynomiale du paramètre n et enfin **exponentielle** quand $f = 2^g$ où g est une fonction polynomiale du paramètre n.

Les complexités (en temps) des algorithmes devront être justifiées.

Rappelons qu'un graphe non-orienté est la donnée (S, A) de deux ensembles finis :

— un ensemble S de **sommets**, et

— un ensemble $A \subset S \times S$ d'**arêtes**, tel que pour tout couple de sommets (s, t), on a $(s, t) \in A$ si et seulement si $(t, s) \in A$. Par convention, on n'admettra pour arêtes que des couples (s, t) avec $s \neq t$.

Soit $G = (S, A)$ un graphe et soit $s \in S$ un sommet de G. Un **voisin** de s est un sommet t de G qui est relié à s par une arête, c'est à dire tel que $(s, t) \in A$. On note $V(s)$ l'ensemble des voisins de s. Le **degré** $d(s)$ de s est le cardinal de $V(s)$. Le **degré** $d(G)$ de G est le maximum des degrés de ses sommets.

Un graphe est dit **étiqueté** lorsque l'on dispose d'une fonction, dite d'étiquetage, de l'ensemble de ses sommets vers un ensemble non vide arbitraire, que l'on appelle ensemble des étiquettes. Les étiquettes peuvent par exemple être des entiers, des listes ou des chaînes de caractères.

On dit qu'une fonction d'étiquetage L est un **coloriage** des sommets de $G = (S, A)$ lorsque deux sommets voisins ont toujours deux étiquettes distinctes (alors appelées **couleurs**), c'est à dire lorsque L vérifie la condition

$$\forall s, t \in S,\ (s, t) \in A \Rightarrow L(s) \neq L(t)$$

Un graphe est dit k-coloriable s'il admet un coloriage avec au plus k couleurs.

Le **nombre chromatique** d'un graphe non orienté G, noté $\chi(G)$, est le nombre minimal k tel que G est k-coloriable. Cet énoncé porte sur le calcul de nombres chromatiques et de coloriages.

On se fixe dans cet énoncé une représentation des graphes par matrices d'adjacence. On se fixe également comme convention que les étiquetages des graphes sont tous à valeurs entières. L'étiquetage d'un graphe sera donné par un tableau d'entiers. On a ainsi

```
type graphe = bool array array
type etiquetage = int array
```

Un graphe non orienté $G = (S, A)$ avec $S = \{0, \ldots, n-1\}$ est représenté par une valeur `gphe` de type `graphe` telle que pour $i, j \in S$, `gphe.(i).(j)=true` si et seulement si $(i, j) \in A$. Le graphe G étant supposé non orienté, on a alors également par symétrie `gphe.(j).(i)=true`. Pour un étiquetage `etiq` de `gphe`, l'étiquette du sommet i de `gphe` et donnée par `etiq.(i)`.

Partie I - Coloriages

1) On considère le graphe `g1` ci-dessous.

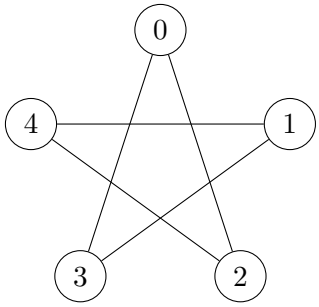


Figure 1 : graphe `g1`

Donner la matrice d'adjacence de ce graphe.

On considère alors les deux étiquetages `etiq1 = [|0;1;2;2;1|]` et `etiq2 = [|0;1;1;2;0|]`. Dire pour chacun si c'est un coloriage du graphe `g1`.

2) Donner le nombre chromatique, ainsi qu'un exemple de coloriage pour le **graphe de Petersen** représenté ci-dessous. On justifiera que le coloriage donné utilise un nombre minimal de couleur.

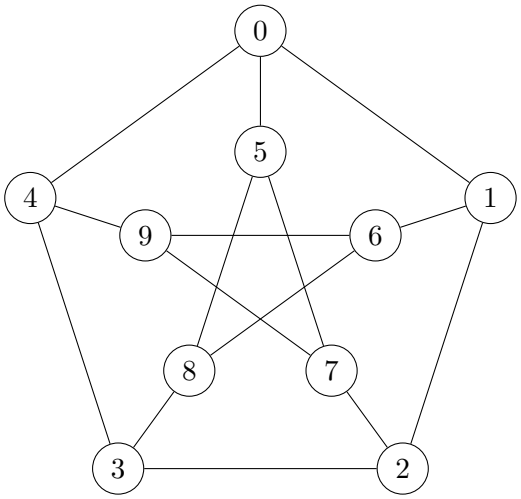


Figure 2 : Le graphe de Petersen de sommets $0, \ldots, 9$

3) Quels sont les graphes G tels que $\chi(G) = 1$?

Soit $n \geq 1$. Justifier que tout graphe à n sommets est n-coloriable et donner un graphe à n sommets tels que $\chi(G) = n$.

4) La vérification de la propriété de coloriage est le problème suivant.
— Entrée : un graphe G et un étiquetage L de G.
— Question : L est-il un coloriage de G ?
Ecrire une fonction

`est_col : graphe → etiquetage → bool`

de complexité quadratique en le nombre de sommets du graphe, telle que `ext_col gphe etiq` renvoie `true` si et seulement si `etiq` est un coloriage de `gphe`. Dans le cas où la taille de l'étiquetage est strictement inférieure au nombre de sommets du graphe, la fonction renvoie `false`.

Il faudra justifier que votre fonction a une complexité quadratique.

5) Soit $k \in \mathbb{N}^*$. Combien il y a-t-il d'étiquetages avec au plus k couleurs d'un graphe G ayant n sommets ?

En déduire que le calcul du nombre chromatique d'un graphe peut s'effectuer en temps exponentiel en le nombre de sommets

Partie II - 2-Coloriages

Nous avons vu à la question 5 que le calcul du nombre chromatique peut s'effectuer en temps exponentiel en le nombre de sommets du graphe. Dans le cas général, on ne sait aujourd'hui pas faire mieux. Pour obtenir de meilleures bornes de complexité, il faut donc se limiter à des sous-probèmes. On considère dans cette partie le cas du 2-coloriage. Il est clair que l'on peut procéder composante connexe par composante connexe du graphe. Nous considérerons donc dans la suite de cette partie que les graphes considérés sont connexes.

Un graphe G est **biparti** lorsque l'ensemble de ses sommets S peut être divisé en deux sous-ensembles disjoints T et U (non vides), tels que chaque arête a une extrémité dans T et l'autre dans U.

6) Démontrer qu'un graphe G est biparti si et seulement s'il est 2-coloriable

On se propose de programmer la vérification de la 2-colorabilité des graphes en procédant comme suit. On effectue un parcours du graphe en profondeur au cours duquel on construit une 2-coloration du graphe. On se donne pour ce faire trois étiquettes, disons -1, 0 et 1. L'étiquette -1 signifiant que le sommet n'a pas encore été considéré et on teste la 2-colorabilité avec 0 et 1. L'étiquetage est donc initialisé à -1 pour tous les sommets . Le principe de l'algorithme est le suivant.

étape 1 : On choisit un sommet quelconque s d'étiquette -1.

étape 2 : On colorie les sommets rencontrés lors du parcours en profondeur à partir de s, en alternant entre les couleurs 0 et 1 à chaque incrémentation de la profondeur, et en vérifiant si les sommets déjà coloriés rencontrés sont d'une couleur compatible.

7) Écrire une fonction

`deux_col : graphe` $\rightarrow$ `etiquetage`

telle que `deux_col gphe` renvoie un 2-coloriage de `gphe` si `gphe` est 2-coloriable. Le coloriage utilisera les couleurs 0 et 1. On demande une complexité quadratique en le nombre de sommets du graphe (et une justification de ce fait). Le comportement de la fonction est laissé au choix du candidat lorsque `gphe` n'est pas 2-coloriable mais il devra être précisé.

Indication : on pourra se donner un étiquetage `etiq` de longueur `Array.length gphe`, dont toutes les cases sont initialisées à -1, et que l'on met à jour au fur et à mesure du parcours de `gphe`.

Partie III - Algorithmes gloutons

Dans cette partie, nous allons étudier deux algorithmes permettant de colorier un graphe en temps polynomial, mais donnant en général un coloriage sous-optimal : le coloriage obtenu peut dans certains cas utiliser plus de couleurs que le coloriage optimal.

Ces deux algorithmes prennent en paramètre un ordre sur les sommets du graphe, que l'on appelera **ordre de numérotation**.

Par exemple, $1 < 3 < 4 < 0 < 2 < 6 < 5 < 9 < 8 < 7$ et $0 < 9 < 7 < 2 < 3 < 6 < 8 < 5 < 1 < 4$ sont deux ordres de numérotation des sommets du graphe de Petersen (figure 2).

Pour un graphe `gphe` à n sommets, on implémente un ordre de numérotation de ses sommets par un tableau `num` de n valeurs entières, tel que `num.(k)=j` si et seulement si le sommet j apparaît en $(k+1)$-ième position dans l'ordre.

Nous commençons par l'**algorithme glouton** de coloriage. Cet algorithme construit un coloriage L d'un graphe G en utilisant au plus $d(G)+1$ couleurs. Son principe est le suivant :

On parcourt la liste des sommets du graphe, dans l'ordre de numérotation des sommets donné.

Pour chaque sommet s parcouru :

étape 1 : On calcule l'ensemble $C(s) = \{L(t)\ ,\ t \in V(s)\}$ des couleurs déjà données aux voisins de s.

étape 2 : On cherche le plus petit entier naturel c qui n'appartient pas à $C(s)$.

étape 3 : On pose $L(s) = c$.

8) Considérons le graphe de Petersen (figure 2) et les deux ordres de numérotation

```
num1=[|1;3;4;0;2;6;5;9;8;7|]
num2=[|0;9;7;2;3;6;8;5;1;4|]
```

Donner les coloriages obtenus par l'algorithme glouton décrit ci-dessus pour le graphe de Petersen et chacun de ces deux ordres de numérotation, ainsi que les nombres de couleurs correspondants.

9) Écrire une fonction

`min_couleur_possible : graphe → etiquetage → int → int`

telle que pour un graphe `gphe` à n sommets, un étiquetage `etiq` à valeurs dans $\{-1, \dots, n-1\}$ (la valeur -1 signifie que le sommet n'a pas encore été colorié), et pour un sommet `s` de `gphe`, l'appel de `min_couleur_possible gphe etiq s` renvoie le plus petit entier naturel n'appartenant pas à l'ensemble $\{\texttt{etiq}.(t)\ ,\ t \in V(\texttt{s})\}$. On demande une complexité $O(n)$.

Indication : on pourra commencer par construire un tableau `coul` *de taille n de booléens tel que pour* `i` *entre 0 et $n-1$,* `coul(i)` *vaut* `true` *si et seulement s'il existe un voisin du sommet* `s` *coloré avec la couleur* `i`*.*

10) À l'aide de la fonction précédente, écrire une fonction

`glouton : graphe → int array → etiquetage`

telle que, pour un graphe `gphe` et un ordre de numérotation `num` de ses sommets, `glouton gphe num` renvoie le coloriage glouton de `gphe`, avec au plus $d+1$ couleurs, où d est le degré de `gphe`. On demande une complexité $O(n^2)$, où n est le nombre de sommets de `gphe`.
On supposera que le tableau `num` contient bien un ordre de numérotation des sommets de `gphe`.

11) Montrer que l'algorithme de coloriage glouton construit toujours un coloriage, et que ce coloriage utilise au plus $d+1$ couleurs, où d est le degré du graphe en entrée.

12) Soit G un graphe. Montrer que pour tout coloriage L de G, il existe un ordre de numérotation des sommets tel que le coloriage glouton L' associé vérifie $L'(s) \leq L(s)$ pour tout sommet s de G. En déduire qu'il existe une numérotation des sommets telle que l'algorithme glouton renvoie un coloriage optimal.

Les questions 8 et 12 indiquent que l'efficacité de l'algorithme glouton est en grande partie dépendante de l'ordre dans lequel on choisit de parcourir les sommets du graphe. L'ordre correspondant à la représentation choisie du graphe (dans notre cas, les indices de la matrice d'adjacence, c'est à dire la permutation identité) est le plus simple à calculer, mais a peu de chances d'être efficace. A contrario, on pourrait essayer de déterminer l'ordre optimal, dont on a prouvé l'existence à la question 12, mais cela n'apporte aucun bénéfice vis-à-vis de la complexité temporelle du problème.
Une alternative est donnée par l'optimisation de Welsh-Powell. L'idée est de parcourir l'ensemble des sommets du graphe par ordre de degré décroissant. Le tri des sommets par degré décroissant ne prend pas plus de temps que le parcours glouton, mais permet d'obtenir un algorithme raisonnablement efficace en pratique.

13) Écrire une fonction

`degres : graphe → int array`

telle que si `gphe` est un graphe ayant n sommets, `degres gphe` renvoie un tableau `t` tel que pour tout i compris entre 0 et $n-1$, `t.(i)` est le degré du sommet i dans le graphe `gphe`. Préciser la complexité de la fonction.

14) Écrire une fonction

`tri_degre : graphe → int array`

qui calcule un tableau des sommets d'un graphe trié par ordre décroissant de leurs degrés (deux sommets ayant le même degré étant placés dans un ordre arbitraire). On expliquera clairement quel algorithme de tri on utilise et on donnera sa complexité.

En déduire une fonction `welsh_powell : graphe → etiquetage` qui implémente l'optimisation de Welsh-Powell. Quelle est la complexité de la fonction `welsh_powell` ?