Composition d'informatique n°1

(Durée : 4 heures)

L'utilisation de la calculatrice **n'est pas autorisée** pour cette épreuve.

⋆ ⋆ ⋆

Graphes de flots

Remarques OCaml :

- On autorisera toutes les fonctions des modules `Array` et `List`, ainsi que les fonctions de la bibliothèque standard (celles qui s'écrivent sans nom de module, comme `max`, `incr` ainsi que les opérateurs comme `@`). Sauf précision de l'énoncé, l'utilisation d'autres modules sera interdite.
- Lorsqu'une question de programmation demande l'écriture d'une fonction de la forme `f (arg1: type1) (arg2: type2) : type3`, cela signifie que la fonction `f` prend pour argument un objet `arg1` de type `type1` et un objet `arg2` de type `type2` et renvoie un objet de type `type3`. Il n'est pas demandé dans l'écriture du code de rappeler ces types. On pourra par exemple commencer l'écriture de la fonction par `let f arg1 arg2 = ... .`
- Lorsque le candidat écrira une fonction, il pourra faire appel à des fonctions définies dans les questions précédentes, même si elles n'ont pas été traitées. Il pourra également définir des fonctions auxiliaires, mais devra préciser leurs rôles ainsi que les types et significations de leurs arguments. Les candidats sont encouragés à expliquer les choix d'implémentation de leurs fonctions lorsque ceux-ci ne découlent pas directement des spécifications de l'énoncé. Si les paramètres d'une fonction à coder sont supposés vérifier certaines hypothèses, il ne sera pas utile dans l'écriture de cette fonction de tester si les hypothèses sont bien satisfaites.

On identifiera une même grandeur écrite dans deux polices de caractères différentes, en italique du point de vue mathématique (par exemple n) et en Computer Modern à chasse fixe du point de vue informatique (par exemple `n`).

Sans précision supplémentaire, lorsqu'une question demande la complexité d'une fonction, il s'agira de la complexité temporelle dans le pire des cas. La complexité sera exprimée sous la forme $\mathcal{O}(f(n,m))$ où n et m sont les tailles des arguments de la fonction, et f une expression la plus simple possible. Les calculs de complexité seront justifiés succinctement.

Préliminaires

Un graphe de flot est une structure permettant de modéliser tout types de flux : réseau de transport routier, circulation d'eau ou de fluides dans des canalisations, distribution d'électricité, ... Sa définition ressemble à celle d'un graphe orienté pondéré, où les poids seront considérés comme des capacités de débit.

On appelle **graphe de flot** un quintuplet $G = (S, A, c, s, t)$ où :

- (S, A) est un graphe orienté sans boucle (c'est-à-dire sans arête de la forme (u, u)) ;
- $c : A \to \mathbb{R}_+^*$ est une fonction dite de **capacité** ;
- il existe deux sommets particuliers : $s \in S$ est une **source** de (S, A), c'est-à-dire un sommet sans arête entrante, et $t \in S$ est un **puits** de (S, A), c'est-à-dire sans arête sortante ;
- s'il existe une arête $(u, v) \in A$, alors $(v, u) \notin A$: il n'existe pas de cycle de taille 2.

La figure 1 représente un graphe de flot G_0.

Si $G = (S, A; c, s, t)$ est un graphe de flot, on appelle **flot de** G une fonction $\phi : A \to \mathbb{R}^+$ vérifiant :

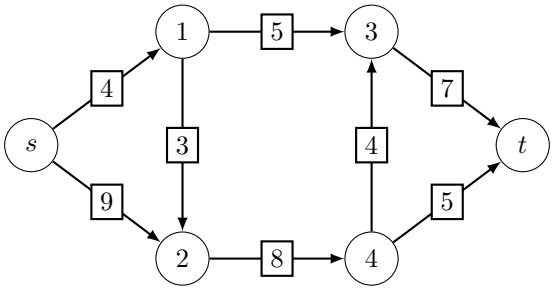

FIGURE 1 – Le graphe de flot G_0.

– respect de la capacité : pour $(u, v) \in S^2$, $\phi(u, v) \leqslant c(u, v)$;
– conservation du flot : pour $u \in S \setminus \{s, t\}$, $\sum_{(v,u)\in A} \phi(u, v) - \sum_{(u,v)\in A} \phi(u, v) = 0$.

Pour faire à nouveau le parallèle avec un transport de liquide dans des tuyaux, l'interprétation de la première règle est que le débit ne peut pas excéder la capacité du tuyau et l'interprétation de la deuxième règle est que ce qui rentre dans une jonction est égal à ce qui sort de cette jonction.

La figure 2 représente le graphe de flot G_0 et un flot ϕ_0 compatible avec G. Pour chaque arête (u, v), on représente sur l'arête $\phi_0(u, v)/c(u, v)$.

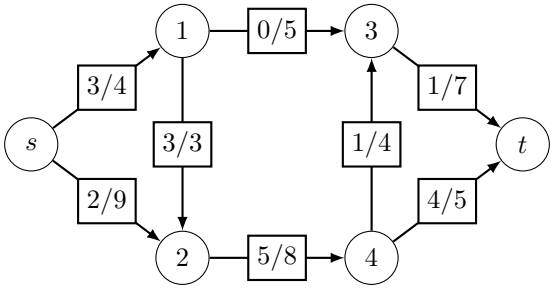

FIGURE 2 – Le graphe de flot G_0 et un flot ϕ_0.

On appelle **débit** de ϕ la valeur $|\phi| = \sum_{(s,u)\in A} \phi(s, u)$, c'est-à-dire la somme des flots sortant de s.

1 Flots

Question 1 Montrer en justifiant succinctement qu'il n'existe pas de flot de G_0 de débit 13.

Pour la suite du sujet, on considère $G = (S, A, c, s, t)$ un graphe de flot et ϕ un flot de G. Si $u \in S$, on appelle :

– **flux entrant de** u la valeur $\phi_-(u) = \sum_{(v,u)\in A} \phi(v, u)$;
– **flux sortant de** u la valeur $\phi_+(u) = \sum_{(u,v)\in A} \phi(u, v)$;
– **flux net de** u la valeur $\Delta\phi(u) = \phi_+(u) - \phi_-(u)$.

Ainsi, la condition de conservation du flot ϕ est équivalente à la propriété suivante :

$$\text{Pour tout } u \in S \setminus \{s, t\}, \Delta\phi(u) = 0$$

Question 2 Montrer que $|\phi| = \Delta\phi(s)$ puis que $\Delta\phi(s) = -\Delta\phi(t)$.

On représente un graphe orienté (S, A) en OCaml par tableau de listes d'adjacence, c'est-à-dire avec le type :

```
type graphe = int list array;;
```

L'ensemble des capacités associées à un graphe de flot $G = (S, A, c, s, t)$ sera représenté par matrice d'adjacence, c'est-à-dire par un objet de type :

```
type capa = float array array;;
```

Ainsi, si $G = (S, A, c, s, t)$ est représenté par un objet `g` de type `graphe` et un objet `c` de type `capa`, alors :

- $S = [\![0, n-1]\!]$, avec $n =$ `Array.length g` est la taille de `g` ;
- par convention, $s = 0$ et $t = n - 1$ sont la source et le puits ;
- pour $u \in S$, `g.(u)` contient la liste des voisins de u, dans un ordre arbitraire ;
- `c` est une matrice de dimensions $n \times n$;
- pour $(u, v) \in S^2$, `c.(u).(v)` $= c(u, v) > 0$ si $(u, v) \in A$ et `c.(u).(v)` $= 0$ sinon.

Par exemple, le graphe de flot G_0 de la figure 1 peut être représenté par les objets :

```
let g0 = [| [1; 2];
            [2; 3];
            [4];
            [5];
            [3; 5];
            [] |];;

let c0 = [| [|0.; 4.; 9.; 0.; 0.; 0.|];
            [|0.; 0.; 3.; 5.; 0.; 0.|];
            [|0.; 0.; 0.; 0.; 8.; 0.|];
            [|0.; 0.; 0.; 0.; 0.; 7.|];
            [|0.; 0.; 0.; 4.; 0.; 5.|];
            [|0.; 0.; 0.; 0.; 0.; 0.|] |];;
```

Comme pour les capacités, on représente un flot ϕ comme une matrice de flottants :

```
type flot = float array array;;
```

Le flot ϕ_0 de la figure 2 peut être représenté par :

```
let phi0 = [| [|0.; 3.; 2.; 0.; 0.; 0.|];
              [|0.; 0.; 3.; 0.; 0.; 0.|];
              [|0.; 0.; 0.; 0.; 5.; 0.|];
              [|0.; 0.; 0.; 0.; 0.; 1.|];
              [|0.; 0.; 0.; 1.; 0.; 4.|];
              [|0.; 0.; 0.; 0.; 0.; 0.|] |];;
```

Pour toute la suite, on supposera que lorsqu'une fonction prend en argument plusieurs tableaux, ces tableaux ont la même dimension et on ne demandera pas de le vérifier.

Question 3 Écrire une fonction `delta_phi (g: graphe) (phi: flot) : float array` qui calcule un tableau `dphi` de taille $n = |S|$ tel que pour $u \in S$, `dphi.(u)` $= \Delta\phi(u)$. On attend une complexité linéaire en $|S| + |A|$.

Question 4 En déduire une fonction `conservation (g: graphe) (phi: flot) : bool` qui renvoie `true` si et seulement si le flot `phi` respecte la règle de conservation du flot dans le graphe de flot représenté par `g`.

Cette règle ne dépend pas des capacités, donc on ne met pas `c` *en argument.*

2 Calcul de flot maximal

On considère le problème d'optimisation `Flot maximal` :

* **Instance** : un graphe de flot $G = (S, A, c, s, t)$.
* **Solution** : un flot ϕ de G.
* **Optimisation** : Maximiser $|\phi|$.

Question 5 Représenter graphiquement une solution à `Flot maximal` sur le graphe G_0 de la figure 1. Justifier qu'il s'agit bien d'une solution maximale.

2.1 Saturation des chemins

On dit qu'une arête $(u, v) \in A$ est **saturée** si $\phi(u, v) = c(u, v)$. On dit qu'un chemin de s à t est **saturé** s'il contient une arête saturée. Le flot ϕ est dit **saturé** si tous les chemins de s à t sont saturés.

Pour un chemin σ de s à t, la **capacité restante** de ce chemin, notée $c_\phi(\sigma)$, est le minimum des valeurs $c(u, v) - \phi(u, v)$ pour (u, v) une arête de ce chemin. Ainsi, un chemin saturé est un chemin de capacité restante nulle. On remarque que pour $u \neq v \in S^2$, (u, v) étant un chemin de longueur 1, la quantité $c_\phi(u, v)$ est bien définie et égale à $c(u, v) - \phi(u, v)$.

Si un chemin σ de s à t n'est pas saturé, on définit l'action de **saturation du chemin σ pour ϕ** comme une modification de ϕ qui consiste, pour chaque arête (u, v) du chemin, à augmenter $\phi(u, v)$ de $c_\phi(\sigma)$. La figure 3 montre le résultat de la saturation du chemin $(s, 2, 4, 3, t)$ dans le graphe G_0 à partir du flot ϕ_0. On a augmenté le flot de 3 le long du chemin. Les arêtes saturées sont $(2, 4)$ et $(4, 3)$.

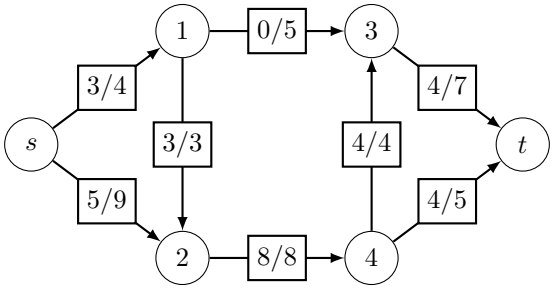

FIGURE 3 – Le graphe de flot G_0 et le flot ϕ_1 après saturation du chemin $(s, 2, 4, 3, t)$ pour le flot ϕ_0.

On considère l'algorithme glouton suivant :

Entrée : Graphe $G = (S, A, c, s, t)$ graphe de flot
Début algorithme
  Poser $\phi(u, v) = 0$ pour tout $(u, v) \in A$.
  Tant que il existe un chemin σ non saturé de s à t **Faire**
    Saturer σ pour ϕ.
  Renvoyer ϕ.

Question 6 Terminer l'exécution de l'algorithme sur le graphe G_0 à partir du flot ϕ_1 de la figure 3. On demande de détailler les chemins non saturés trouvés et leur capacité restante. On représentera graphiquement le résultat.

2.2 Algorithme de Ford-Fulkerson

La question précédente et la question `qflotmax` montrent que cet algorithme glouton ne renvoie pas toujours un flot maximal. Pour corriger le problème, il faut envisager de pouvoir faire « refluer » le flot en arrière dans une arête.

Pour $G = (S, A, s, t, c)$ un graphe de flot et ϕ un flot de G, on définit le **graphe résiduel** $G_\phi = (S, A_\phi, r_\phi)$ comme un graphe pondéré orienté tel que $(u, v) \in A_\phi$ si et seulement si l'une des deux conditions est vérifiée :

- $(u, v) \in A$ et $\phi(u, v) < c(u, v)$, auquel cas on pose $r_\phi(u, v) = c_\phi(u, v) = c(u, v) - \phi(u, v)$;
- ou $(v, u) \in A$ et $\phi(v, u) > 0$, auquel cas on pose $r_\phi(u, v) = \phi(v, u)$.

Le premier type d'arêtes correspond aux arêtes de G non saturées, auxquelles on associe une nouvelle capacité correspondant à la capacité restante. Le deuxième type d'arêtes correspond aux arêtes retournées de G dont le flot traversant est strictement positif ; la nouvelle capacité correspond à la quantité de flot qu'on peut faire refluer (faire revenir en arrière).

Attention, le graphe G_ϕ peut contenir des arêtes n'existant pas dans G. Par ailleurs, ce n'est a priori pas un graphe de flot, car il n'a pas forcément de source ou de puits.

Question 7 Représenter graphiquement le graphe résiduel de G_0 avec le flot ϕ_1 représenté figure 3. On représentera les valeurs de r_ϕ comme des capacités d'un graphe de flot.

Question 8 Écrire une fonction `residuel (g: graphe) (c: capa) (phi: flot) : graphe` qui construit la partie graphe (S, A_ϕ) d'un graphe résiduel pour un flot `phi` dans un graphe de flot représenté par `g` et `c`. Quelle est la complexité temporelle de cette fonction ?

On appelle **chemin améliorant pour** ϕ un chemin de s à t dans le graphe résiduel G_ϕ.

Question 9 Écrire une fonction `parcours (g: graphe) : int array` qui calcule l'arborescence d'un parcours en profondeur dans un graphe orienté (S, A), en partant du sommet 0. Le résultat prendra la forme d'un tableau `parent` tel que :

- `parent.(0)` vaut 0 ;
- si u est accessible depuis 0, alors `parent.(u)` est le parent de u dans l'arborescence ;
- si u n'est pas accessible depuis 0, alors `parent.(u)` vaut -1.

On garantira une complexité $\mathcal{O}(|S| + |A|)$ et on demande de justifier brièvement cette complexité.

Question 10 Écrire une fonction `chemin (parent: int array) : int list option` qui prend en argument un tableau `parent` décrivant une arborescence de parcours telle que décrite à la question précédente et renvoie une option de liste telle que :

- s'il n'existe pas de chemin de s (le sommet 0) à t (le sommet $n - 1$), alors la fonction renvoie `None` ;
- sinon, la fonction renvoie `Some sigma` où `sigma` est une liste décrivant un chemin σ de s à t (donc commençant par 0 et terminant par $n - 1$).

La **capacité résiduelle** d'un chemin améliorant σ, qu'on notera $r_\phi(\sigma)$, correspond au minimum des $c'(u, v)$ pour (u, v) une arête de ce chemin.

Question 11 Écrire une fonction `residu (sigma: int list) (c: capa) (phi: flot) : float` qui calcule la capacité résiduelle d'un chemin σ étant donné une liste décrivant ce chemin et des matrices décrivant la capacité et le flot.

L'action d'**amélioration de ϕ par rapport à** σ est une modification de ϕ qui consiste, pour chaque arête (u, v) du chemin, à modifier ϕ selon le principe suivant :

- si $(u, v) \in A$ et $c(u, v) \geqslant \phi(u, v) + r_\phi(\sigma)$, alors on ajoute $r_\phi(\sigma)$ à $\phi(u, v)$;
- sinon, on retire $r_\phi(\sigma)$ à $\phi(v, u)$.

L'algorithme de Ford-Fulkerson est alors le suivant :

Entrée : Graphe $G = (S, A, c, s, t)$ graphe de flot
Début algorithme
  Poser $\phi(u, v) = 0$ pour tout $(u, v) \in A$.
  Tant que Il existe un chemin améliorant σ pour ϕ **Faire**
    Améliorer ϕ par rapport à σ.
  Renvoyer ϕ.

Question 12 Terminer l'exécution de l'algorithme de Ford-Fulkerson sur le graphe G_0 avec le flot ϕ_1 représenté figure 3. On représentera les graphes résiduels et flots intermédiaires, et on donnera les chemins améliorants et leurs capacités résiduelles. On pourra repartir du flot obtenu à la question 6.

Question 13 Écrire une fonction `ameliorer (sigma: int list) (c: capa) (phi: flot) : unit` qui améliore un chemin supposé améliorant `sigma` pour un flot `phi`. La fonction ne devra rien renvoyer.

On rappelle que la commande `Array.make_matrix m n x` permet de créer une matrice de dimension $m \times n$ dont toutes les cases sont égales à x.

Question 14 En déduire une fonction `ford_fulkerson (g: graphe) (c: capa) : flot` qui renvoie un flot maximal selon l'algorithme de Ford-Fulkerson.

Question 15 Montrer que si les capacités sont entières, l'algorithme de Ford-Fulkerson termine toujours et déterminer sa complexité temporelle en fonction de $|S|$, $|A|$ et le débit du flot maximal $|\phi^*|$.

3 Flot maximal, coupe minimale

Soit $G = (S, A, c, s, t)$ un graphe de flot. On appelle **coupe** de G un ensemble $X \subseteq S$ tel que $s \in X$ et $t \in \overline{X}$. La **capacité** d'une coupe X est $C(X) = \sum_{(u,v) \in A \cap (X \times \overline{X})} c(u, v)$.

Si ϕ est un flot pour G, alors le **flux** d'une coupe X est $\phi(X) = \sum_{(u,v) \in A \cap (X \times \overline{X})} \phi(u, v) - \sum_{(v,u) \in A \cap (\overline{X} \times X)} \phi(v, u)$, c'est-à-dire la différence entre la somme des flots qui sortent de X et la somme des flots qui rentrent dans X.

Question 16 Dans le graphe G_0 donné figure 1, déterminer la capacité de la coupe $X = \{s, 1, 3\}$.

Question 17 Soit ϕ un flot et X une coupe de G. Montrer que $\phi(X) \leqslant C(X)$, puis que $|\phi| = \phi(X)$.

Indication : on pourra montrer que si $x \in X \setminus \{s\}$, alors $\phi(X) = \phi(X \setminus \{x\})$.

Question 18 Montrer l'équivalence entre les propriétés suivantes :

1. ϕ est un flot maximal ;
2. il n'existe pas de chemin améliorant pour ϕ ;
3. il existe une coupe X telle que $|\phi| = C(X)$.

Indication : on pourra s'intéresser à l'ensemble des sommets accessibles depuis s dans G_ϕ.

Question 19 En déduire que s'il termine, l'algorithme de Ford-Fulkerson renvoie un flot maximal.

4 Résolution du problème de couplage maximum

On veut montrer qu'on peut résoudre le problème de recherche de couplage maximum dans un graphe biparti en utilisant l'algorithme de Ford-Fulkerson. On considère un graphe $G = (S = X \sqcup Y, A)$ biparti et non orienté, c'est-à-dire tel que $A \subseteq \{\{x, y\} \mid x \in X, y \in Y\}$.

On définit le **réseau de** G comme un graphe de flot $R_G = (S', A', s, t, c)$ tel que :

- $S' = S \cup \{s, t\}$, où s et t sont deux nouveaux sommets ;
- A' contient les arêtes suivantes :
  - les (s, x) pour $x \in X$;
  - les (y, t) pour $y \in t$;
  - les (x, y) pour $\{x, y\} \in A$.
- Toutes les capacités des arêtes sont égales à 1.

Par exemple, la figure 4 représente un graphe biparti G_1 et son réseau associé.

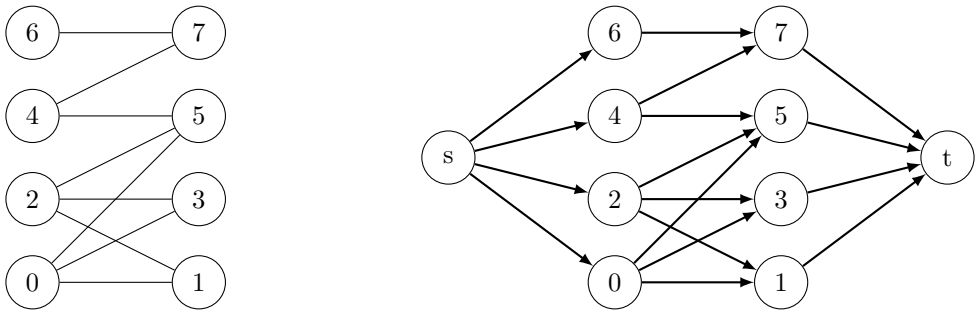

FIGURE 4 – Le graphe biparti G_1 et le réseau R_{G_1}. Les capacités ne sont pas représentées car égales à 1.

Question 20 Expliquer comment résoudre le problème du couplage maximum en utilisant l'algorithme de Ford-Fulkerson dans R_G.

Question 21 Justifier la correction et déterminer la complexité temporelle de l'algorithme précédent.

5 Algorithme d'Edmonds-Karp

On souhaite améliorer la complexité de l'algorithme de Ford-Fulkerson dans le pire cas, qui peut être très désavantageux si on choisit les mauvais chemins améliorants.

Question 22 Montrer que le nombre de passages dans la boucle **Tant que** de l'algorithme de Ford-Fulkerson peut être égal à 2 ou à 2000 selon le choix des chemins améliorants dans le graphe de flot G_2 représenté figure 5.

L'algorithme d'Edmonds-Karp est une variante de l'algorithme de Ford-Fulkerson qui consiste à choisir un **plus court** chemin améliorant à chaque itération.

5.1 Implémentation de file

On souhaite dans cette partie implémenter une structure de file pour l'utiliser pour l'écriture d'un parcours en largeur. On utilise pour cela une structure de liste simplement chaînée, en gardant en mémoire le maillon de début et le maillon de fin. Pour faciliter la manipulation d'une file vide, on ajoute un maillon particulier dit **sentinelle**.

La figure 6 représente schématiquement la structure, pour une file contenant les valeurs $5, 12, 7$ et 3.

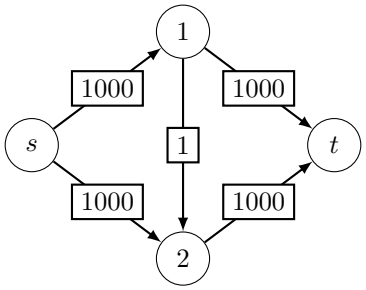

FIGURE 5 – Le graphe de flot G_2.

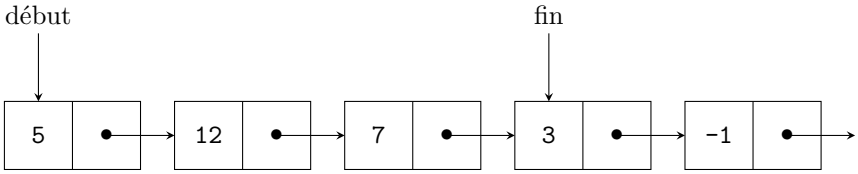

FIGURE 6 – Une liste simplement chaînée avec marquage du début et de la fin. Le dernier maillon est la sentinelle.

On implémente un maillon par un objet du type :

```
type maillon = {valeur : int;
                mutable suivant : maillon};;
```

tel que si `m` est de type `maillon`, alors `m.valeur` représente la valeur du maillon, et `m.suivant` représente le maillon suivant (sauf pour le cas particulier de la sentinelle).

On définit la sentinelle par :

```
let rec sentinelle = {valeur = -1; suivant = sentinelle};;
```

On ne demande pas de comprendre cette syntaxe utilisant `rec` pour autre chose qu'une fonction, et on pourra utiliser la variable `sentinelle` comme si elle était définie de manière globale.

On implémente une file par un objet du type :

```
type file = {mutable debut : maillon;
             mutable fin : maillon};;
```

tel que si `f` est de type `file`, alors :

- si la file correspondante est vide, `f.debut` et `f.fin` sont tous les deux égaux au maillon `sentinelle` ;
- sinon, `f.debut` est égal au premier maillon de la liste et `f.debut` est égal au dernier maillon de la liste, c'est-à-dire celui qui précède le maillon sentinelle.

Pour toute la suite, on supposera que les valeurs contenues dans la file sont des valeurs entières positives ou nulles.

Question 23 Quelle opération est problématique en termes de complexité si on choisit d'enfiler des éléments en les rajoutant au début de la liste et de défiler en supprimant à la fin de la liste ?

Pour la suite, on enfilera des éléments à la fin de la liste et on défilera au début de la liste.

Question 24 Écrire une fonction `creer () : file` qui crée une file vide.

Question 25 Écrire une fonction `est_vide (f: file) : bool` qui teste si une file est vide.

Question 26 Écrire une fonction `enfiler (f: file) (x: int) : unit` qui enfile un élément dans une file. On prendra garde à bien gérer le cas où la file est initialement vide.

Question 27 Écrire une fonction `defiler (f: file) : int` qui défile et renvoie un élément dans une file. La fonction renverra une erreur si la file est déjà vide.

Question 28 Quelle est la complexité temporelle des opérations précédentes ?

5.2 Algorithme d'Edmonds-Karp

L'algorithme d'Edmonds-Karp est le suivant :

Entrée : Graphe $G = (S, A, c, s, t)$ graphe de flot
Début algorithme
  Poser $\phi(u, v) = 0$ pour tout $(u, v) \in A$.
  Tant que Il existe un chemin améliorant pour ϕ **Faire**
    Poser σ un **plus court** chemin améliorant.
    Améliorer ϕ par rapport à σ.
  Renvoyer ϕ.

La recherche de plus court chemin peut se faire avec l'algorithme de parcours en largeur.

Question 29 Écrire une fonction `parcours_largeur (g: graphe) : int array` qui calcule l'arborescence d'un parcours en largeur dans un graphe orienté (S, A), en partant du sommet 0. Le résultat prendra la forme d'un tableau `parent` comme défini à la question 9.

Dans la suite de cette partie, on cherche à déterminer la complexité de l'algorithme précédent.

On note $\phi_0, \phi_1, \ldots, \phi_k$ la suite des flots à chaque itération de la boucle **Tant que** lors de l'exécution de l'algorithme d'Edmonds-Karp, et $\sigma_1, \ldots, \sigma_k$ les plus courts chemins améliorants correspondants. Pour $i \in [\![0, k]\!]$ et $u \in S$, on note $d_i(u)$ la **distance** de s à u dans le graphe résiduel G_{ϕ_i}.

On commence par montrer que la longueur des chemins σ_i ne peut qu'augmenter avec i.

Question 30 On suppose pour cette question qu'il existe $i \in [\![1, k-1]\!]$ et un sommet v dans le chemin σ_{i+1} tel que $d_i(v) > d_{i+1}(v)$. On pose v un sommet qui minimise $d_{i+1}(v)$ parmi ceux qui vérifient cette propriété. On pose u le prédécesseur de v dans σ_{i+1}.

Par une disjonction de cas, montrer qu'on arrive à une contradiction.

On note $|\sigma_i|$ la longueur d'un chemin σ_i. On appelle **arête critique** de σ_i une arête (u, v) de G_{ϕ_i} telle que $r_{\phi_i}(u, v) = r_{\phi_i}(\sigma_i)$.

Question 31 Montrer qu'une arête (u, v) ne peut être critique qu'au plus $\frac{|S|}{2}$ fois au cours de l'algorithme.

Question 32 En déduire la complexité temporelle de l'algorithme d'Edmonds-Karp en fonction de $|S|$ et $|A|$, en supposant $|A| \geqslant |S|$.

⋆ ⋆ ⋆