- Graphe orienté

Un graphe orienté pondéré simple peut être représenté en mémoire par sa matrice d'adjacence. Dans le travail de programmation Julia à rendre, vous utiliserez le type présenté ci-contre dans lequel :

```
mutable struct tabgraph
   nv::Int
   ne::Int
   adj::Array{Bool,2}
   pds::Array{Real,2}
   pred::Array{Int,2}
   tabgraph(nbs::Int) =
     new(nbs,0,
       zeros(Bool,(nbs,nbs)),
       Array{Real}(undef,nbs,nbs),
       Array{Int}(undef,nbs,nbs))
end
```

- `nv` désigne le nombre de sommets du graphe (*vertices* en anglais) ;
- `ne` désigne le nombre d'arêtes (*edges* en anglais) ;
- `adj[s,d]` précise si une arête existe du sommet s vers le sommet d (les sommets sont numérotés à partir de 1) ;
- `pds[s,d]` est alors le poids de cette arête (sa valeur n'a pas d'importance et ne doit pas être considérée si l'arête n'existe pas) ;
- `pred` permettra d'utiliser la même structure pour mémoriser des chemins, `pred[s,d]` désignant le sommet d'où venir pour arriver à `d` en venant de `s`.
- `tabgraph(nbs)` est un constructeur qui permet de créer un graphe à `nbs` sommets ne comportant aucune arête.

1 - Utilitaires

En vous inspirant du code ci-contre qui retourne une copie d'un graphe , écrivez les fonctions indispensables pour utiliser ce type :

```
import Base.copy

function copy(G::tabgraph)
   F=tabgraph(G.nv)
   F.ne=G.ne
   F.adj=copy(G.adj)
   F.pds=copy(G.pds)
   F.pred=copy(G.pred)
   return F
end
```

- `relie!(G,s,d,p)` qui ajoute une arête de poids `p` du sommet `s` au sommet `d` ou modifie son poids si cette arête existe déjà ;
- `aff(G)` qui montre le graphe sous un format lisible par un être humain, très utile pendant le débogage ;
- `alea!(G)` qui remplit aléatoirement un graphe ;
- `voisins(G,s)` qui retourne l'ensemble des sommets `d` pour lesquels une arête de `s` à `d` existe dans `G` ;
- `lire!(G,nomf)` qui complète le graphe `G` à partir d'un fichier texte (de nom `nomf`) dont chaque ligne comporte une arête : sommet de départ, sommet d'arrivée, poids, les lignes comportant un sommet non présent dans G doivent être ignorées ;
- et d'autres si vous avez des idées.

2 - Fermeture transitive

Écrivez une fonction `connexitéforte(G::tabgraph)::tabgraph` retournant un nouveau graphe dont la matrice d'adjacence donne pour chaque couple `(s,d)` de sommets la présence d'un chemin de `s` à `d` dans `G`. Cette fonction ne doit pas modifier `G`. Une définition de la fermeture transitive est accessible sur [wikipedia](). Dans cette quesion, on en se soucie pas des poids, le graphe retourné doit avoir une matrice de poids remplie de 1.

Remarquez que si une arête de `s` à `d` est ajoutée pour chaque triplet `(s,z,d)` tels qu'une arête existe de `s` à `z` et une de `z` à `d`, alors la matrice d'adjacence caractérisera tous les sommets joignables en une ou deux arêtes. En répétant suffisamment le processus, les `adj[s,d]` seront vrais si et seulement s'il existe un chemin de `s` à `d` dans `G`. Cette méthode est appelée [algorithme de Warshall]().

Écrivez ensuite une autre version de la même fonction avec le moins possible de boucles explicites, en utilisant le fait que le produit de deux matrices d'adjacence constituées de 1 et de 0 au lieu de booléens fournit l'existence de chemins composés d'une arête du premier graphe et d'une arête du second. D'ailleurs Julia considère dans certains cas les booléens comme des entiers 0 ou 1 (Bool hérite de Integer). Utilisez aussi la notation pointée (*broadcast*).

Comparer les temps d'exécution des deux programmations sur des graphes aléatoires.

3 - Plus courts chemin

L'algorithme de Warshall peut se généraliser à des arêtes de poids non unitaires. C'est l'[algorithme de Roy-Floyd-Warshall](). La méthode est la même, chaque `pds[s,d]` doit être remplacé par `pds[s,m]+pds[m,d]` si cette valeur est plus petite ou que les sommets n'étaient pas encore reliés.

Écrire le code correspondant et le tester.

