# TP 29

## Algorithmes sur les graphes

# 1 Graphes au format dimacs

## A. Le module Scanf

Le module `Scanf` permet de *scanner* des chaînes de caractères. On va utiliser ici la fonction `sscanf` (la répétition du `s` n'est pas une coquille) de ce module. Cette fonction prend en argument :

- la chaîne de caractères à scanner ;
- une chaîne de caractères de formattage à scanner (identique à la chaîne de formattage utilisé pour la fonction `Printf.printf`) ;
- une fonction d'arité égale au nombre de valeurs à lire (donc décrit dans la chaîne précédent).

**Exemple.**

```
# let s = "val1: 42 val2: 41 val3: 2\n" in
Scanf.sscanf s "val1: %d val2: %d val3: %d\n" (fun x y z -> x * y / z)
;;
- : int = 861
```

Il faut imaginer ici que la chaîne `s` a pu être récupérée par la lecture de l'entrée standard ou la lecture d'un fichier.

## B. Lecture de fichier

On rappelle ici ce qui avait vu dans le TP 20 sur les tries et les anagrammes pour lire un fichier en OCaml :

- `open_in : string -> in_channel` qui prend en argument un nom de fichier (c'est-à-dire un chemin vers un fichier) et renvoie une valeur de type `in_channel` ;
- `input_line : in_channel -> string` qui lit des caractères dans le `in_channel` jusqu'à tomber sur un retour à la ligne, et renvoie la chaîne de caractères lue (sans le `'\n'`) ;
- `close_in : in_channel -> unit` pour fermer le fichier une fois qu'on a fini de l'utiliser.

Il y a deux remarques importantes :

- une valeur de type `in_channel` se comporte comme un « flux », c'est-à-dire que les lignes sont « consommées » au fur et à mesure qu'elles sont lues (si l'on appelle deux fois de suite `input_line` sur le même `in_channel`, on obtiendra une ligne puis la suivante) ;
- si on appelle `input_line` sur un `in_channel` « épuisé » (dans lequel il n'y a plus rien à lire), l'exception `End_of_file` est levée. C'est en fait le seul moyen de savoir (en utilisant ces fonctions) qu'on a terminé la lecture du fichier.

## C. Le format dimacs

Le format *dimacs* est un format standard de description de graphes sous forme d'un fichier texte. Les fichiers fournis sont très simples :

- toute ligne commençant par un caractère `c` est un commentaire et doit être ignorée (ces lignes peuvent apparaître n'importe où dans le fichier) ;
- il y a une unique ligne commençant par `p edge` et contenant ensuite deux entiers (séparés par une espace) ; ces deux entiers indiquent respectivement le nombre de sommets et le nombre d'arêtes du graphe ;
- les autres lignes sont de la forme `e i j` (le caractère `e` suivi de deux entiers), et indiquent la présence d'une arête reliant le sommet *i* et le sommet *j*. Les lignes de cette forme apparaissent toutes après la ligne `p edge ...`

**Exemple.**

```
c Graphe cycle à 4 sommets
p edge 4 4
e 1 2
e 1 4
e 2 1
e 2 3
e 3 2
e 3 4
e 4 1
e 4 3
```

**Remarques.**

- Je considère ici que le format *dimacs* décrit des graphes *a priori* orientés (ça n'est pas forcément standard pour ce type de fichier). Ainsi, pour modéliser un graphe non orienté, comme dans l'exemple précédent, les deux lignes `e i j` et `e j i` seront nécessaires pour coder l'arête *ij* ;
- les sommets sont numérotés à partir de 1 !

*Exercice 29.1* Écrire une fonction `lire_dimacs` qui prend en entrée un nom de fichier au format *dimacs* et renvoie le graphe correspondant.

```
lire_dimacs : string -> graph
```

## D. Pour les tests

Vous trouverez des fichiers `.gr` sur le site, ainsi que des tests, pour vérifier vos fonctions.

# II Accessibilités et composantes connexes

*Exercice 29.2*

Écrire une fonction `accessible` telle que l'appel `accessible g x y` détermine si $y$ est accessible depuis $x$ dans le graphe (*a priori* orienté) $g$.

```
accessible : graph -> vertex -> vertex -> bool
```

On fera en sorte que cette fonction réponde dès que possible (le plus simple est d'utiliser ici une exception).

*Exercice 29.3*

1. Écrire une fonction `tab_composantes` qui prend en entrée un graphe *supposé non orienté* et renvoie un tableau $t$ tel que $t_i = t_j$ si et seulement si les sommets $x_i$ et $x_j$ sont dans la même composante connexe du graphe.

   ```
   tab_composantes : graph -> int array
   ```

2. Écrire une fonction `tab_composantes` qui prend en entrée un graphe *supposé non orienté* et renvoie le nombre de composantes connexes du graphe.

   ```
   nb_composantes : graph -> int
   ```

# III Acyclicité

*Exercice 29.4* On travaille avec des graphes donnés sous forme d'un tableau de listes d'adjacence.

1. Écrire une fonction OCaml `est_dag` qui détermine si un graphe orienté est acyclique.

   ```
   est_dag : graph -> bool
   ```

2. Écrire une fonction `cycle` prenant en entrée un graphe orienté et renvoyant :
   - `None` si le graphe ne possède pas de cycle ;
   - `Some u`, où $u$ est une liste de sommets du graphe formant un cycle orienté, si le graphe possède un cycle.

   ```
   cycle : graph -> vertex list option
   ```

3. Quelle difficulté supplémentaire lors de la vérification de l'acyclicité apparaît dans un graphe *non orienté*. Comment peut-on alors faire ?

# IV Tri topologique

*Exercice 29.5* Écrire une fonction `tri_topologique` renvoyant un tri topologique du graphe direct acyclique passé en argument, sous la forme d'une liste.
On lèvera l'exception `Cycle` si un tel tri n'existe pas.

```
exception Cycle
tri_topologique : graph -> vertex list
```

# V Bonus

## A. Princess in a castle

A princess inhabits a flight [1] of 17 rooms in a row. Each room has a door to the outside, and there is a door between adjacent rooms. The princess spends each day in a room that is adjacent to the room she was in the day before.
One day a prince arrives from far away to woo [2] for the princess. The guardian explains the habits of the princess and also the rules to him : each day he may knock at an outside door of his choice. If the princess is behind it she will open and in the end marry him. If not, nothing happens, and he gets another chance the next day.
Unfortunately his return ticket expires after 30 days. Does he have enough time to conquer the princess ?

## B. Princess on a graph

*What if the n rooms are laid out in any arrangement, not necessarily a line ?*

1. ça peut vouloir dire « étage » en anglais
2. courtiser