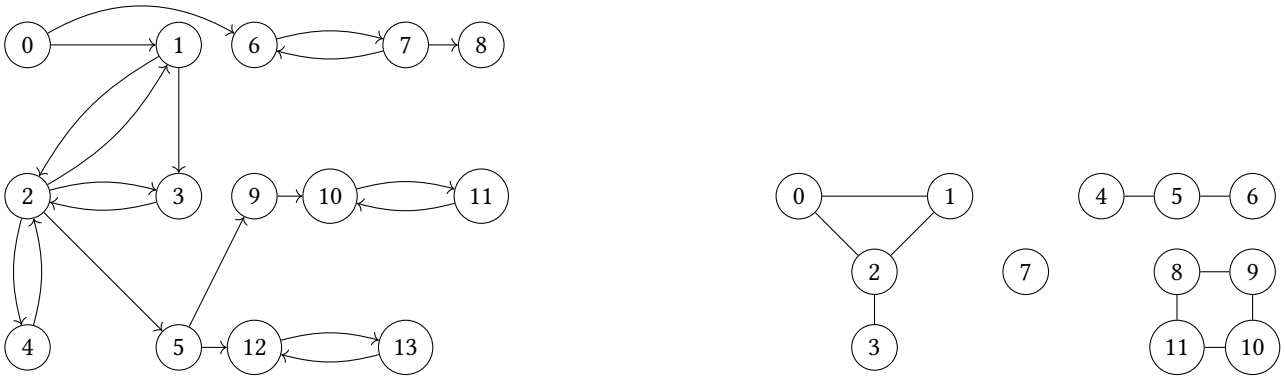
TRAVAUX PRATIQUES XXI

Apprendre à marcher (sur un graphe)

Le but de ce TP est d'implémenter les algorithmes usuels de parcours de graphe : parcours en profondeur, en largeur, recherche de composantes connexes, tri topologique et détection de cycle.

Dans tout ce TP, on pourra utiliser les modules Queue et Stack de OCaml.

On indice les sommets par $[\![0; n[\![$ et représente les graphes par **listes d'adjacences**.



(a) g0 (b) g2

FIGURE XXI.1 – Deux exemples du cours

1. Encoder les deux exemples ci-dessus par liste d'adjacence, en OCaml. On appelera les deux variables globales créées g0 et g2. *Chacun d'entre eux doit être encodé en moins de 3 minutes.*

Dans la suite, on posera `let g1 = desoriente g0` aka une version non-orientée de g0.

De plus, pour les parcours, on donne des couleurs aux sommets du graphe en fonction de leur statut dans le parcours :

— Un sommet qui a déjà été visité est dit `Noir`.

— Un sommet qui a été ajouté à la structure de données d'attente mais n'a pas encore été visité est dit `Gris`.

— Un sommet qui n'a pas été ajouté dans la structure de données est dit `Blanc`.

```
1 type etat = Blanc | Gris | Noir
```

On pourra au choix dans ce TP utiliser ces couleurs ou des booléens pour différencier les sommets déjà vus des autres. L'avantage de ces couleurs est qu'elles prennent en compte les deux moments possibles où l'on peut vouloir marquer un sommet comme vu, qui diffère selon le type de parcours qu'on souhaite effectuer : en largeur ou en profondeur.

Les fonctions de ce TP sont à savoir faire et refaire les yeux fermés !

Théorème 1 OCaml : parcours.ml

A Outils pratiques de OCaml

A.1 `List.map` et `List.iter`

Il arrive souvent que l'on souhaite appliquer une fonction à toute une liste. Soit `l = [l1; ...; ln]` une liste. Deux cas de figure :

— Si on veut calculer la liste `[f(l1); ...; f(ln)]`, on utilise la fonction suivante :
`List.map : ('a -> 'b) -> 'a list -> 'b list` .
`List.map f l` renvoie `[f l1; ...; f ln]` c'est à dire la liste des images par `f` des éléments de `l` .

— Si on veut simplement appliquer une fonction qui renvoie le type `unit` à tous les éléments de la liste (par exemple pour afficher la liste), on utilise [1] :
`List.iter : ('a -> unit) -> 'a list -> unit` .
`List.iter f l` évalue `f` sur chacun des éléments de `l` . C'est donc équivalent à `let _ = List.map f l`.

1. Utiliser `List.iter` pour faire une fonction qui affiche une liste d'entiers.

A.2 type `'a option`

Parfois, on veut faire une fonction qui renvoie une valeur particulière « si elle réussit à la trouver », et ne renvoie rien sinon.

Il existe un type somme pré-défini parfait pour ça : le type `'a option` :

OCaml
```
type 'a option = None | Some of 'a
```

2. Écrire une fonction `array_in_index : 'a -> 'a array -> int option` qui recherche `x` dans un tableau `t` et renvoie un `int option` contenant soit un indice `i` tel que `T.(i)` vaille `x` soit `None` s'il n'y a pas de tels `i`.

B Parcours en largeur (Breadth-First Search)

On définit un type `etat` ainsi :

OCaml
```
type etat = Blanc | Gris | Noir
```

On rappelle qu'un sommet est dit Blanc s'il n'est jamais entré dans la structure d'attente du parcours, Gris s'il y est entré mais pas encore ressorti, et Noir sinon.

Quand vous implémenterez les parcours, au lieu de simplement mémoriser un marquage `deja_vu`, vous mémoriserez *l'état* de chaque sommet (sa couleur). Autrement dit, vous maintiendrez un tableau `etat` tel que `etat.(i)` soit l'état actuel du sommet `i` : Noir, Gris ou Blanc.

N'oubliez pas que vous avez défini deux graphes en début de TP (un orienté, et un non orienté), que vous pouvez utiliser pour vos tests de parcours !

Le but de cette section est de programmer l'algorithme de parcours en largeur vu en cours, en OCaml. Nous en avons déjà donné un code complet en OCaml, le but est d'en écrire progressivement une version légèrement différente, en commençant par plus simple pour ajouter les éléments petit à petit. Il vous est donc demandé de :

1) commencer par essayer d'écrire les fonctions sans consulter le cours
2) au besoin, consulter le pseudo-code très général d'un parcours
3) en désespoir de cause, consulter le code OCaml du parcours en largeur, **PUIS le ranger**, et écrire la fonction sur son fichier sans le regarder à nouveau

Vous ne devez en aucun cas recopier sans comprendre le code donné dans le cours !

Pour implémenter une file, vous pouvez utiliser les fonctions suivantes (j'appelle `'a file` le type des files dont le contenu est `'a` ; mais son véritable nom est `'a Stack.t`) :

— `Queue.create : unit -> 'a file` crée une file (qui sera un objet mutable, comme les tableaux).

— `Queue.is_empty : 'a file -> bool` renvoie `true` SSI la file passée en argument est vide.

— `Queue.add : 'a -> 'a file -> unit` prend en argument un élément et une file et modifie cette file (‼) pour ajouter cet élément à l'entrée de la file.
`Queue.push` est un autre nom de cette fonction.

— `Queue.take : 'a file -> 'a` prend en argument une file, la modifie (‼) pour en sortir un élément (celui qui est le prochain à sortir), et renvoie cet élément.
`Queue.pop` est un autre nom de cette fonction.

— `Queue.peek : 'a file -> 'a` renvoie la prochain élément à sortir d'une file mais ne la modifie pas. `Queue.top` en est un autre nom.

— Pour encore plus de fonctions : cf documentation du module Queue.

1. On pourrait techniquement utiliser `List.map` , mais la liste des `f li` est juste une liste de `()`.

3. Écrire une fonction `bfs_affiche : graphe -> sommet -> unit` qui prend en argument un graphe `g` et un sommet `s` et effectue un parcours en largeur de `g` depuis `s`. Le traitement d'un sommet consiste à afficher son étiquette. Vous le ferez de deux façons :
   a. En considérant un sommet comme étant déjà vu lorsqu'il est Noir (i.e. maruage à la sortie de la file).
   b. En considérant un sommet comme étant déjà vu lorsqu'il est Gris (i.e. marquage à l'entrée de la File).
   c. Quelle différence y a-t-il ? Commentez ?

   Aide : Vous pourrez définir une fonction auxiliaire `ajoute_voisin` dans la boucle, et la donner en argument à List.iter. Alternativement, vous pouvez aussi définir directement une fonction auxiliaire récursive pour parcourir la liste des voisins (comme dans le cours), mais je vous conseille de travailler avec List.iter, c'est plus simple et plus direct, et ça vous forcera à ne pas écrire la même chose que dans le cours.
4. Modifiez votre fonction précédente pour qu'elle applique un traitement quelconque sur les sommets (pas forcément un affichage), sous la forme d'une fonction donnée en argument. Son nouveau type sera :
   `bfs : (sommet -> unit) -> graphe -> sommet -> unit` où `bfs traitement g s` parcours en largeur `g` depuis `s` en appliquant `traitement` à chaque sommet.

L'intérêt du parcours en largeur est de calculer des distances et des plus courts chemins. On va vouloir construire l'arbre des plus courts chemin depuis une source `s0`. On va se servir pour cela du tableau `pred` du pseudo-code donné en cours. Quand on enfile un sommet voisin, on ajoute l'arête/arc correspondant :on ajoute un arc/arête de u vers v lorsque l'exploration de u est responsable du fait que v est vu pour la première fois.

5. a. Modifiez votre bfs pour qu'il renvoie le tableau `pred` des prédécesseurs dans le parcours. On posera `pred.(s0) = s0`.
      Vous noterez qu'il s'agit exactement d'une représentation d'un arbre par tableau de parenté.
   b. Vérifiez sur des exemples !
6. a. En modifiant votre fonction précédente ou en en créant une nouvelle, construisez de même le tableau `dist` qui retient la distance d'un sommet vu à la source. `dist.(i)` est la distance du sommet `i` à la source `s0` du parcours.
   b. Déduisez-en une méthode pour construire une matrice `dist_all` telle que `dist_all.(i).(j)` soit la distance $d(\texttt{i}, \texttt{j})$ de `i` à `j`. Quelle est sa complexité ? Implémentez-la.

Enfin, on peut utiliser des parcours en largeur pour rechercher des composantes connexes. L'idée est en fait de faire plusieurs parcours : chaque parcours identifie exactement une composante connexe.

7. Écrire une fonction `comp_connexe` qui prend en entrée un graphe et renvoie une représentation de ses composantes connexes.
   La question est volontairement ouverte : plusieurs représentations sont possibles. Aide : vous pouvez par exemple construire une liste de listes (chacune des listes étant une composante connexe), ou bien un tableau indexé par les sommets dont la case d'indice i contient le représentant (sommet source s0) de la composante connexe qui contient le sommet i.

C Parcours en profondeur (Depth-First-Search) et applications

Le fichier `decouvertes.ml` fourni avec ce TP contient un graphe orienté étiqueté par les sommets. On en donne une représentation par liste d'adjacences dans la variable `ladja`. Vous pouvez accéder à cette variable depuis votre fichier du TP par la syntaxe `Decouvertes.ladja` .

Le module `Decouvertes` contient également une variable `noms` : le tableau de ses étiquettes. Autrement dit, le sommet `i` a pour liste de voisins `Decouvertes.ladja.(i)` et pour étiquette `Decouvertes.noms.(i)`.

C'est un graphe de « découvertes technologiques », qui indique des dépendances imaginaires entre elles. Il y a un arc de i vers j si connaitre i permet de découvrir j. Par exemple, il y a un arc de 1 (économie) vers 65 (entreprise) : connaitre l'économie permet de découvrir l'entreprise. *Ce graphe n'a aucune prétention à une quelconque rigueur historique.*

Nous allons travailler dessus à l'aide de parcours en profondeur.

8. Trouvez toutes les découvertes qui n'ont pas de pré-requis, c'est à dire les sommets de degré entrant 0.
9. Sur une feuille, avec un papier et un crayon, rappelez-vous les deux façons d'écrire un parcours en profondeur (itératif avec une pile et récursif). Dans les questions suivantes, sauf indication contraire, prenez la version avec laquelle vous êtes plus à l'aise.
10. Le sommet d'indice 7 représente les Mathématiques. Affichez à l'aide d'un dfs :
    a. Toutes les découvertes qui sont descendantes des mathématiques, c'est à dire pour lesquelles il est nécessaire d'avoir découvert les mathématiques.

b. Toutes les découvertes qui ne sont pas descendantes des mathématiques, c'est à dire pour lesquelles il n'est pas nécessaire d'avoir découvert les mathématiques.

11. Écrire une fonction générale `dfs : (sommet->'a) -> (sommet->unit) -> graphe -> sommet -> unit` qui telle que `dfs pre post g s` effectue un parcours en profondeur **récursif** de `g` depuis `s` en appliquant à chaque sommet la fonction `pre` quand on l'ouvre (quand on commence son appel récursif) et la fonction `post` quand on le ferme (quand on a fini de traiter ses enfants). Testez-la.

Vous pouvez par exemple utiliser les fonctions suivantes pour ouvrir et fermer :

OCaml : parcours.ml

```
let ouvre : sommet->unit =
  Printf.printf "Ouvre␣sommet␣%d\n"
let ferme : sommet->unit =
  Printf.printf "Ferme␣sommet␣%d\n"
```

On peut représenter le statut d'un sommet dans un parcours récursif (ouvert, fermé ou pas encore abordé) par le type suivant :

OCaml

```
type ouverture = Vierge | Ouvert | Ferme
```

D Pour occuper les plus rapides

12. Proposer une fonction pour trouver, dans un graphe non-orienté, le plus long chemin élémentaire entre deux sommets x et y. Quelle est sa complexité ?
(Si vous avez une complexité polynomiale, prouvez que votre code est faux.)