TRAVAUX PRATIQUES XXIII

Apprendre à marcher (sur un graphe)

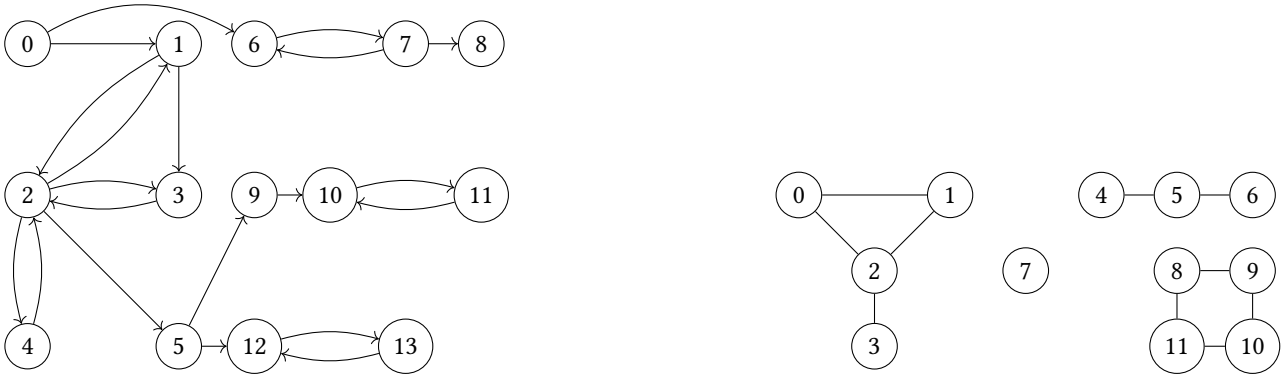
Le but de ce TP est d'implémenter les algorithmes usuels de parcours de graphe : parcours en profondeur, en largeur, recherche de composantes connexes, tri topologique et détection de cycle.

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 Préambule

A.1 Listes d'adjacence



(a) `g0` (b) `g2`

FIGURE XXIII.1 – Deux exemples du cours

0. 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 peut être encodé en moins de 3 minutes. Au boulot !*

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

A.2 Fonctions pratiques en OCaml

Dans ce TP, on sera beaucoup amené·es à utiliser les `Option` et la fonction `List.iter`. Si vous avez des questions, c'est le moment !

A.3 Boucle `while`

Comme les boucles `for`, les boucles `while` existent en OCaml. Ce sont elles aussi des expressions de type `unit` :

```
(* ceci est une expression *)
while booleen do
  expr_a_evaluer_a_chaque_iter
done
```

B Parcours en largeur (Breadth-First Search)

Débutons par le parcours en largeur. Nous aurons besoin de files : le module `Queue` en fournit. Ce sont des files impératives, dont le type s'appelle `Queue.'a t` mais pour plus de lisibilité je le noterai `'a file` ; et qui contient notamment les fonctions ci-dessous :

- `Queue.create : unit -> 'a file` crée une file (mutable).
- `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 y enfiler l'élément.
- `Queue.take : 'a file -> 'a` prend en argument une file, la modifie pour en défiler un élément et renvoie cet élément.
- `Queue.peek : 'a file -> 'a` renvoie la prochain élément à sortir d'une file mais ne la modifie pas.
- Pour encore plus de fonctions : cf documentation du module Queue.

1. Écrire une fonction `bfs : (sommet -> unit) -> graphe -> sommet -> unit` telle que `bfs visite g s0` parcours en largeur `g` depuis `s` en appliquant `visite` à chaque sommet.
   Par exemple, `bfs (fun n -> print_int n; print_char ' ') g s0` soit afficher dans l'ordre les sommets visités par un BFS de `g` depuis `s0`
   Aide : lorsque vous en êtes à l'étape "pour chaque voisin, faire..." utilisez un `List.iter`*; vous devrez sans doute commencer par définir la fonction à itérer.*

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 veut calculer un tableau `parent` tel que :

- Si `u` a un parent `p` dans l'arbre de parcours, alors `parent.(u) = Some p`
- Sinon, `parent.(u) = None`

Pour cela, on peut par exemple insérer des arcs dans la file.

2. Modifiez votre bfs pour qu'il renvoie le tableau `parent` des prédécesseurs dans le parcours.
3. Comment modifier le bfs pour qu'au lieu d'un tableau de parenté, il renvoie le tableau des distances depuis `s0` ? On pourra discuter de comment représenter $+\infty$.

C Calcul des composantes connexes

À l'aide du parcours de votre choix, on peut calculer les composantes connexes. Pour rappel, l'idée est de faire un parcours complet : chacun des parcours identifie exactement une composante connexe.

4. Écrire une fonction `comp_connexe` qui prend en entrée un graphe et renvoie une représentation de ses composantes connexes.

La question ci-dessus est volontairement ouverte : je ne décris pas comment sont représentées les composantes. Vous pouvez par exemple :

- Faire un tableau qui à un sommet associe le numéro de sa composante.
- Faire un tableau qui à un sommet associe un sommet particulier de sa composante, appelé le *représentant.*
- Faire la liste des composantes, qui sont elles-mêmes une liste de sommets.

D Parcours en profondeur (Depth-First-Search)

D.1 Sur les exemples du cours

5. Écrire une fonction `dfs : graphe -> sommet -> unit` qui parcourt un arbre en profondeur et affiche les ouvertures et fermeture successives. On pourra utiliser les fonctions `ouvre` et `ferme` fournies pour ces affichages.
6. À l'aide de `dfs`, faire de même mais lors d'un parcours complet.

D.2 Sur le graphe des découvertes technologiques

Le fichier `civ.ml` fourni avec ce TP contient un graphe orienté étiqueté par les sommets. Sa liste d'adjacence est `Civ.decouvertes` . Les sommets sont étiquettés par des noms donnés dans `Civ.noms` : en fait, ce graphe est un graphe de progrès scientifiques (issu du jeu Civilization II, aucune prétention à la réalité historique). Il y a un arc d'un sommet u vers un sommet v si la connaissance scientifique du sommet u est nécessaire pour découvrir le sommet v; par exemple il y a un arc du sommet « économie » vers le sommet « entreprise ». Cette relation de nécessité est transitive.

Les noeuds du graphe sont appelés des *technologies*.

7. Trouvez toutes les découvertes qui n'ont pas de pré-requis, c'est à dire les sommets de degré entrant 0.
8. 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 celles dont le noeud est accessible depuis le noeud des mathématiques.
   b. Toutes les découvertes qui ne sont pas descendantes des mathématiques.
9. En utilisant votre fonction `dfs` , créez une fonction `dates` qui prend en argument un graphe, en effectue un parcours en profondeur **récursif complet** et renvoie deux tableaux contenant respectivement les dates d'ouverture et de fermeture des sommets.

On veut maintenant faire un tri topologique, c'est à dire toruver un ordr epossible de découverte des technologies.

10. Écrire une fonction `tri_topologique : graphe -> sommet list` qui prend en argument un DAG et renvoie une liste de ses sommets triés dans un ordre topologique.

D.3 Retour aux exemples du cours

On peut déclarer (globalement) une exception nommée `Cycle` comme suit :

```
exception Cycle
```

Pour lever l'exception Cycle (« déclencher l'erreur »), on utilise l'expression `raise Cycle`

11. Modifier le tri topologique pour que si le graphe donné en entrée n'est pas un DAG, la fonction lève l'exception `Cycle`.
12. Testez sur `g0` et sur le graphe des technologies !

E Pour occuper les plus rapides

13. Écrire une fonction `est_arbre : graphe -> bool` qui teste si un graphe non-orienté est un arbre, c'est à dire s'il est connexe et sans cycle.
14. 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 en la taille du graphe, prouvez que votre code est faux.)