TP n°30 - Graphes - Parcours en profondeur (DFS) - Applications

Dans ce TP, on implémente un algorithme de parcours en profondeur d'un graphe (*Depth-First Search* DFS an anglais) de manière récursive et itérative, puis, on montre le lien avec l'ordre post-fixe vu sur les arbres. Enfin, on montre deux applications :

- résolution d'un problème d'ordonnancement de tâches (tri topologique)
- calcul de composantes connexes d'un graphe non-orienté
- calcul de composantes fortement connexes d'un graphe orienté

Pour réaliser ce TP, un module OCaml de graphe pondéré `Wgraph.ml` est mis à votre disposition sur Moodle. Il contient - entre autres - les primitives de manipulation d'un graphe pondéré :

`init_no_edge: int -> bool -> 'a Wgraph.t :` constructeur

`number_of_vertices: 'a Wgraph.t -> int :` accesseur

`number_of_edges: 'a Wgraph.t -> int :` accesseur

`is_directed: 'a Wgraph.t -> bool`

`print: 'a Wgraph.t -> unit :` accesseur

`add_edge: 'a Wgraph.t -> int -> int -> 'a -> unit :` ce transformateur a été adapté par rapport au précédent TP et ne renvoie rien, pour alléger un peu le code. En effet, si nous avions conservé l'implémentation pour laquelle la fonction renvoie un booléen, la syntaxe OCaml nous aurait alors obligé à récupérer cette valeur, ce qui nous aurait obligé à écrire pour chaque ajout d'arc :

```
let _ = add_edge i j w in ();;
```

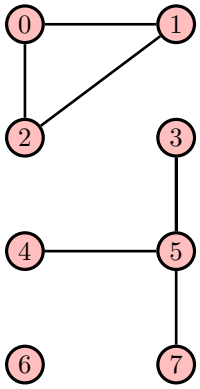
Pour l'utiliser en mode interactif dans Tuareg, il suffit d'écrire, au début de votre fichier :

```
#use ''Wgraph.ml''
```

Toutes les fonctions et tous les tests seront implémentés dans un fichier OCaml nommé `NOM_dfs.ml`
Ce fichier de code devront être déposés sur Moodle pour mercredi prochain.

Exercice 1 (Parcours en profondeur récursif).

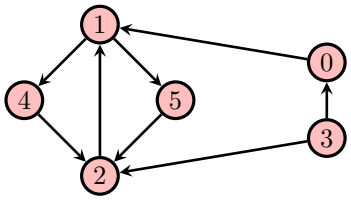
1. Créer le graphe g_1 suivant :



2. Réécrire sur papier, en pseudo-code, de manière propre (**étape obligatoire**) l'algorithme de parcours en profondeur d'un graphe sous forme **récursive**.
3. Implémenter le parcours en profondeur à partir d'un sommet source sous la forme d'une fonction **récursive** `dfs_rec: 'int Wgraph.t -> int -> unit`. Cette fonction se contente d'afficher l'étiquette entière de chaque sommet une fois que tous les chemins partant de ce sommet ont été traités (on dira alors que le sommet est *terminé*). On pourra utiliser la fonction `List.iter` du module `List`.
4. Valider votre fonction sur le graphe g_1, en prenant différents sommets comme source.
5. Y-a-t-il unicité du parcours en profondeur ? Expliquer.
6. Quelle est la complexité temporelle de votre implémentation ?
7. Quelle est la complexité spatiale de votre implémentation ?

Exercice 2 (Calcul des composantes connexes d'un graphe non orienté).

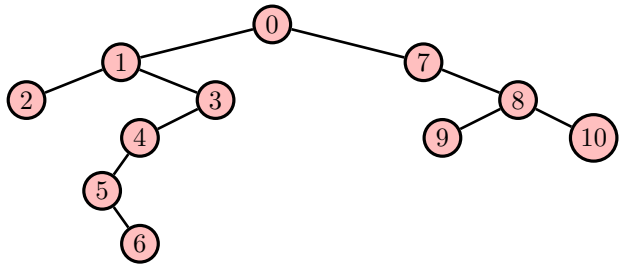
1. Copier votre fonction `dfs_rec` et renommez-la `connected_components`. Adapter cette fonction pour qu'elle renvoie les composantes connexes d'un graphe non-orienté sous la forme d'un tableau de taille n_v, qui associe, à chaque sommet, un entier représentant la couleur de la composante connexe à laquelle il appartient. Le prototype de la fonction sera : : `connected_components: 'a Wgraph.t -> int array *int`. On commencera toujours notre exploration en commençant par le sommet d'étiquete 0. Lorsque l'on a aura exploré l'intégralité d'une composante connexe, on recherchera un nouveau sommet source en choisissant celui de plus petit indice parmi les sommets non visités.
2. Tester votre fonction sur le graphe g_1.
3. Créer le graphe g_2 suivant :



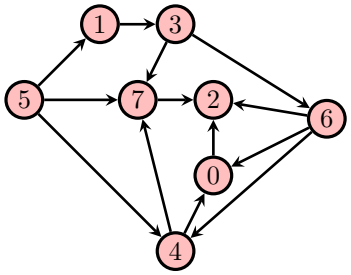
puis tester la fonction `connected_components`. Qu'en pensez-vous ?

Exercice 3 (Ordonnancement de tâches et tri topologique).

1. Copier la fonction `connected_components` et la renommer `dfs_postorder`. Adapter cette nouvelle fonction pour qu'elle renvoie la liste des étiquettes des sommets ordonnées par ordre de fin de traitement, le premier élément de la liste correspondant à l'étiquette du sommet qui a été terminé en dernier. Tester sur les graphes g_1 et g_2. On pourra, au choix, utiliser des listes mutables ou une pile mutable avec le module `Stack`. *Attention : sommet terminé $\neq$ sommet visité $\neq$ sommet sans successeur. Faites bien ma différence.*
2. Quel lien et quelle différence faites-vous avec le parcours post-fixe tel que nous l'avons défini sur les arbres binaires ? Vérifier cette affirmation en testant votre fonction sur le graphe suivant, vu comme un arbre enraciné. Comment expliquer les différences observées ?



3. Ce parcours peut-être utile dans le cadre d'un ordonnancement de tâches dépendant les unes des autres. On représente l'ensemble des tâches comme les sommets d'un graphe et on indique qu'une tâche u doit être effectuée avant une tâche v en créant un arc orienté $u \to v$. Voici un exemple de graphe g_3 de tâches.



4. E créant ce graphe g_3 et en utilisant votre code, indiquez dans quel ordre effectuer les tâches pour que toutes les tâches nécessaires à l'accomplissement d'une tâche T soient effectuées avant T. Y a-t-il unicité de la solution ? Indiquer un autre ordonnancement possible.
5. A quel condition peut-on ordonnancer un graphe de tâches ?

Exercice 4 (Parcours en profondeur itératif à l'aide d'une pile).

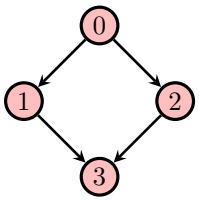
Nous avons vu que l'implémentation récursive de l'algorithme de parcours en profondeur pouvait poser un problème de complexité spatiale dans le segment de pile, avec un possible débordement dans le cas de chemins très longs dans de grands graphes.

1. Écrire, en pseudo code, l'algorithme de parcours en profondeur de manière itérative, en utilisant une pile.
2. Implémenter cet algorithme en OCaml sous la forme d'une fonction `dfs_iter` ayant la même spécification que votre fonction `dfs_rec` du début de ce TP.
3. Comparer les résultats de parcours entre les fonctions `dfs_rec` et `dfs_iter`.

Exercice 5 (Recherche de cycles dans un graphe).

Copier la fonction `dfs_iter` et la renommer `dfs_iter_has_cycle`.

1. **On considère dans un premier temps uniquement le cas d'un graphe non orienté.** Quelle test simple permet de savoir si l'on a trouvé un cycle dans un tel graphe ? Quel cas particulier doit être géré ? Comment faire cela en pratique dans la version itérative ? *Indication : il n'existe pas de cycle de longueur 2 dans un graphe non-orienté....*
2. Modifier la fonction pour repérer et afficher les cycles d'un graphe non-orienté. La fonction renvoie la valeur `true` si au moins un cycle a été trouvé et affiche tous les cycles trouvés. Tester votre fonction sur le graphe g_1 avec différentes sources et sur l'arbre.
3. On considère maintenant le cas des graphes orientés. L'algorithme précédent marche-t-il toujours pour les graphes orientés ? On pourra s'intéresser au test suivant :



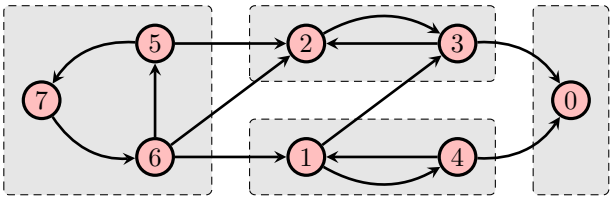
4. Formuler en français le bon critère pour repérer un cycle dans un graphe orienté.
5. Adapter votre fonction pour qu'elle repère la présence de cycles dans un graphe orienté et affiche tous les cycles. *Indication : on pourra créer un nouveau type somme et modifier le type du tableau `visited` pour différencier les sommets jamais visités, les sommets situés sur le chemin en cours d'exploration et les sommets traités.*

Exercice 6 (Calcul des composantes fortement connexe d'un graphe orienté - Algorithme de Kosaraju-Sharir).

Le calcul des composantes fortement connexes de g peut se faire naïvement en testant l'existence d'un chemin dans le graphe pour chaque couple de sommets. Cette approche effectuant un parcours complet du graphe à chaque départ de sommet est trop coûteuse en temps. **D'autres approches, comme celle de Kosaraju-Sharir, font le même calcul en temps linéaire en la taille du graphe**. L'algorithme comporte deux étapes.

— Il effectue un parcours en profondeur récursif de g et relève, pour chaque nœud visité v_i, l'instant t_i de *fin de traitement* du nœud. Cet instant se situe à la fin de l'exécution de la fonction de parcours sur le nœud v_i, après le retour des appels récursifs sur ses successeurs dans le graphe.

— Il effectue un deuxième parcours en profondeur, cette fois du graphe g' obtenu en renversant le sens de toutes les arêtes de g. Les différents sommets de départ du parcours sont choisis non pas dans un ordre arbitraire comme dans l'étape précédente mais dans l'ordre décroissant des instants t_i. Pour chaque sommet de départ v_i, l'ensemble des sommets visités par le parcours en profondeur de g' à partir de v_i forme une composante fortement connexe du graphe initial g.



1. On considère le graphe g_4 ci-dessus. On choisit le sommet d'étiquette 0 comme sommet de départ d'un parcours en profondeur.

a. En visitant les successeurs par ordre croissant d'indice, montrer que les instants de fin de traitement des sommets du graphe g_4 de la figure ci-dessus lors d'un parcours en profondeur sont dans l'ordre suivant.

$$t_0 < t_4 < t_2 < t_3 < t_1 < t_6 < t_7 < t_5 \tag{2}$$

b. En déduire la liste des sommets qui permet la mise en œuvre de la second étape de l'algorithme.

c. En déduire les composantes fortement connexes de g_4 dans l'ordre où l'algorithme les construit.

2. Pour effectuer la seconde étape de l'algorithme, il faut d'abord renverser le graphe. Écrire une fonction `reverse: 'a Wgraph.t -> 'a Wgraph.t` qui prend en entrée un graphe et qui renvoie un autre graphe dans lequel les sommets sont les mêmes et le sens de toutes les arêtes est inversé. La complexité de la fonction doit être linéaire en la taille du graphe fourni en entrée.

3. Écrire une fonction `strongly_connected_components: 'a -> int array *int` qui renvoie un tableau qui associe à chaque sommet sa composante fortement connexe ainsi qu'un entier indiquant le nombre de composantes fortement connexes.

4. Vérifier les résultats obtenus sur le graphe g_2 et sur le graphe g_4.