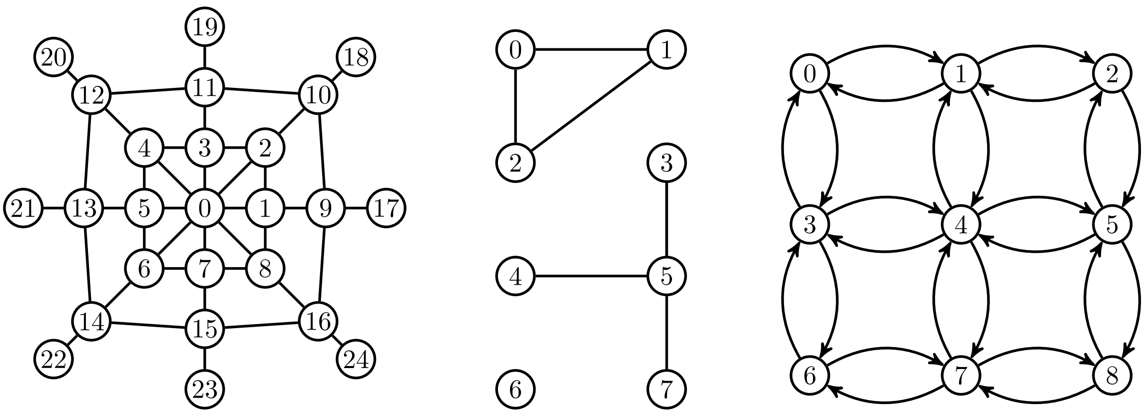
TD - Parcours de graphes

Ce sujet implémente des graphes étiquetés par des entiers, à l'aide de *dictionnaires*, et les *algorithmes de parcours*.

Question 1. Le module `Hashtbl`[1] est adopté pour définir le type `graph`. Proposer une implémentation des fonctions de manipulation nommées ci-dessous. Celles préfixées avec `di` s'appliquent à des graphes *orientés* uniquement. Les autres s'appliquent à des graphes orientés et non orientés.

```
type graph = (int , int list) Hashtbl.t
val vertex_exist :      graph -> int -> bool (* existence d'un sommet *)
val edge_exist :        graph -> int -> int -> bool (* existence d'une arête *)
val vertex_neighbors : graph -> int -> int list (* voisins d'un sommet *)
val edge_add :          graph -> int -> int -> unit (* ajout d'une arête *)
val edge_remove :       graph -> int -> int -> unit (* suppression d'une arête *)
val di_edge_add :       graph -> int -> int -> unit
val di_edge_remove :    graph -> int -> int -> unit
val vertex_add :        graph -> int -> unit (* ajout d'un sommet *)
val vertex_remove :     graph -> int -> unit (* suppresion d'un sommet *)
```

Question 2. Définir trois dictionnaires `g1`, `g2`, `g3` qui représentent respectivement les graphes G_1, G_2, G_3. Les voisins d'un sommet seront stockés dans l'ordre croissant de leurs étiquettes dans la liste.



(a) Graphe G_1 **(b)** Graphe G_2 **(c)** Graphe G_3

Figure 1 – Graphes exemples

Question 3. Le *parcours en profondeur* d'un graphe g d'ordre n à depuis un sommet s est naturellement *récursif*. Un tableau `visited` de n booléens, initialisé à `false`, est mis à jour chaque fois qu'un sommet est visité. Un second tableau `pred` de taille n associe son prédécesseur à chaque sommet dans le parcours.
Le *parcours en largeur* utilise une file. Le sommet de départ y est d'abord ajouté. Tant que la file n'est pas vide, on en extrait un sommet non visité. Après traitement de ce sommet, on le marque comme visité et on ajoute ses voisins dans la file. Le module `Queue` peut être utilisé pour définir une file[2].
Écrire deux fonctions `dfs` et `bfs` qui, après des parcours DFS et BFS respectivement, renvoient le tableau des précédesseurs d'un sommet de départ. Estimer leurs complexités.

```
val dfs : graph -> int -> int array
val bfs : graph -> int -> int array
```

Question 4. S'il existe, on cherche désormais un chemin reliant deux sommets d'un graphe. Comment exploiter les deux fonctions précédentes pour construire un tel chemin ? Quel parcours détermine un plus court chemin dans un graphe non pondéré ?

Question 5. Écrire une fonction `t_comp : graph -> int array` qui reçoit un graphe *non orienté* et qui renvoie un tableau t tel que $t_i = t_j$ si x_i et x_j sont dans la même composante connexe. Par exemple `t_comp g2` peut renvoyer `[|0; 0; 0; 3; 3; 3; 6; 3]`.

Question 6. Sans utiliser la fonction précédente, écrire une fonction `lst_comp : graph -> int list list` qui renvoie une liste de listes des composantes connexes. Par exemple `lst_comp g2` peut renvoyer `[[6]; [7; 4; 5; 3]; [2; 1; 0]]`.

1. https://v2.ocaml.org/api/Hashtbl.html
2. https://v2.ocaml.org/api/Queue.html