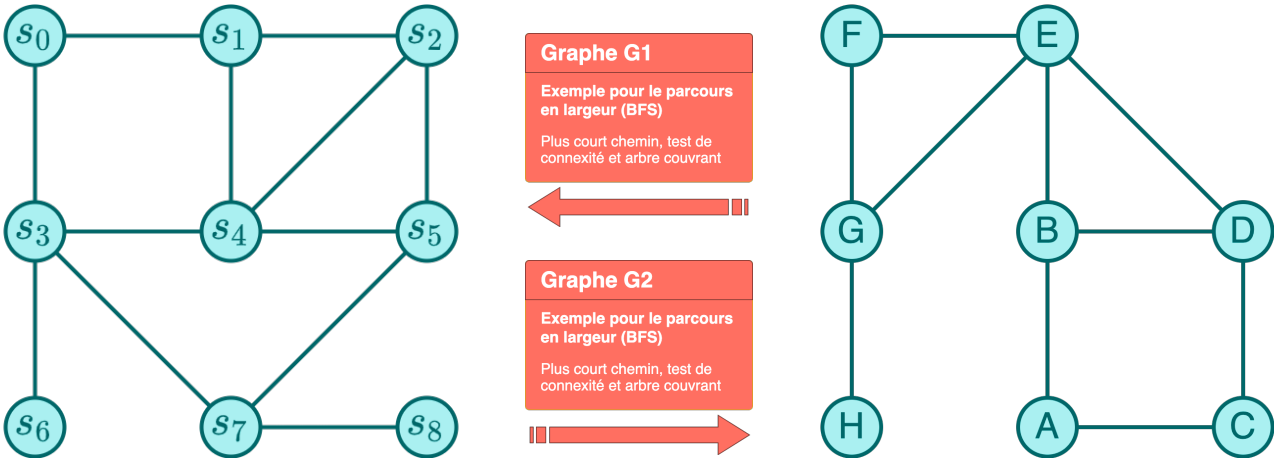
Parcours en largeur (BFS)

ITC – PCSI^2 2024-2025 – Lycée Fabert (Metz) – L. Parise – N.Wipf – S.Limal – D.Mengel

Dans toute la suite du TP, à fins d'exemples, nous étudierons les deux graphes représentés ci-dessous :



0. Compléter les dictionnaires `G1` et `G2` pour qu'ils représentent respectivement les graphes ci-dessus.
1. On rappelle ici l'algorithme de parcours en largeur (BFS) vu en cours :

Algorithme 1 Parcours en largeur

Fonction BFS(G : **graphe**, depart :**sommet**)
Rôle : Effectuer un parcours en largeur en affichant les sommets rencontrés
Déclaration : v, s : **sommet**, attente : **file** # File des sommets en cours en traitement
Début

```
1:  attente ← CreerFileVide()
2:  Initialisation de statut (0 pour chaque sommet)
3:  enfiler(attente, depart)
4:  marquage à 1 du sommet depart dans statut
5:  tant que la file attente est non vide faire
6:      s ← defiler(attente) ;
7:      pour pour tout voisin v de s dans G faire
8:          si si v non marqué dans statut alors
9:              marquage à 1 du sommet v dans statut
10:             enfiler(attente, v)
11:         fin si
12:     fin pour
13:     marquage à 2 du sommet s dans statut
14:     Afficher "Sommet traité :",s
15: fin tant que
```

Fin Fonction

Écrire la fonction `BFS(G:dict,depart:str)→None` qui implémente l'algorithme précédent. Effectuer des essais avec les dictionnaires `G1` et `G2` représentant respectivement les graphes G_1 et G_2.

2. Jusqu'à présent, la fonction `BFS` s'est contentée d'afficher les différents sommets traités lors du parcours en largeur. Il peut toutefois être utile de stocker quelques informations relatives à ce parcours.
   — L'ordre de visite des sommets peut être stocké dans une liste `ordre` (un sommet est ajouté à cette liste au moment où il est retiré de la file d'attente).
   — Un dictionnaire `pere` peut stocker le prédécesseur de chaque sommet lors du parcours en largeur : pour chaque sommet `s` du graphe, `pere[s]` renvoie ainsi le sommet qui a permis de le découvrir. Au début du parcours, on choisit d'initialiser à `None` la valeur renvoyée par `pere[s]` pour chaque sommet `s` du graphe. La valeur de `pere[s]` est actualisée lorsque le sommet `s` est effectivement découvert.

   Modifier la fonction `BFS` pour qu'elle renvoie la liste `ordre` et le dictionnaire `pere` sans effectuer d'affichage. Effectuer des tests sur les graphes G_1 et G_2 : on choisira comme sommet de départ le sommet `"s1"` pour le graphe G_1 et le sommet `"b"` pour le graphe G_2.

3. Sachant que le parcours en largeur permet d'explorer les sommets par distances croissantes depuis le sommet `depart`, ce type de parcours permet d'identifier UN/LE plus court chemin (en nombre d'arêtes) entre les sommets `depart` et `arrivee`.

   Écrire sur ce principe une fonction

   ```
   plusCC(G:dict, depart : str, arrivee:str)→chemin:list
   ```

   qui renvoie la liste `chemin` des sommets permettant de passer de `depart` à `arrivee`, le cas échéant, lors du parcours en largeur (c'est un plus court chemin entre `depart` et `arrivee`) et la liste vide s'il n'y a aucun chemin reliant ces deux sommets.

4. Si on dispose de la liste `ordre` décrite à la question 2, et qui contient tous les sommets d'un graphe G découverts en partant d'un sommet donné, on peut affirmer que G **est connexe si et seulement si il y a autant d'éléments dans `ordre` que de sommets dans** G, ce qui signifie que tous les sommets de G sont dans `ordre` !

   Écrire la fonction `connexite(G:dict)→bool` qui détermine si un graphe (non orienté) est connexe ou non