Graphes et Algorithmes

Travaux Pratiques n°3

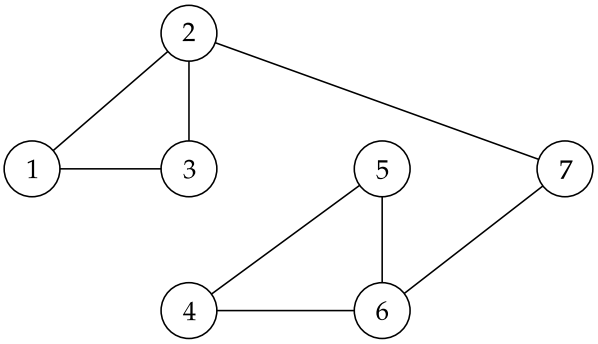
Université de la Nouvelle-Calédonie, 2024

Encore un peu de graphes

Exercice 1

Rappel : Une composante connexe d'un graphe non orienté est un sous-graphe tel qu'il existe un chemin entre tout couple de sommets du sous-graphe. De plus, dans un graphe, un point d'articulation est un sommet dont le retrait entraîne l'augmentation du nombre de composantes connexes tandis qu'un isthme (aussi appelé un pont) est une arête dont le retrait entraîne l'augmentation du nombre de composantes connexes.

Écrire une fonction `is_connected` qui renvoie `True` si le graphe en paramètre est connexe, sinon `False`. Puis, écrire une fonction `nb_connected` qui renvoie le nombre de composantes connexes du graphe en paramètre. Testez ces fonctions sur le graphe non orienté suivant (vous pourrez tester la fonction avec un graphe de type dictionnaire) :



Déterminez les points d'articulation du graphe via la fonction `articulation` que vous aurez implémentée. Déterminez également les isthmes via la fonction `isthme` que vous aurez implémentée.

Exercice 2

Écrire une fonction récursive `has_cycle` qui détecte si un graphe orienté contient un cycle depuis le noeud en paramètre. Vous pourrez utiliser le graphe orienté du TP1 pour vérifier votre fonction.

Indication : Pendant le parcours du graphe, si vous trouvez un noeud qui a déjà été visité et qui n'est pas le parent direct du noeud courant, alors il y a un cycle.

Écrire une fonction `detect_cycle` qui détecte si un graphe orienté contient un cycle depuis n'importe quel noeud.

À cheval entre les graphes et les arbres : L'algorithme de Prim

Exercice 1

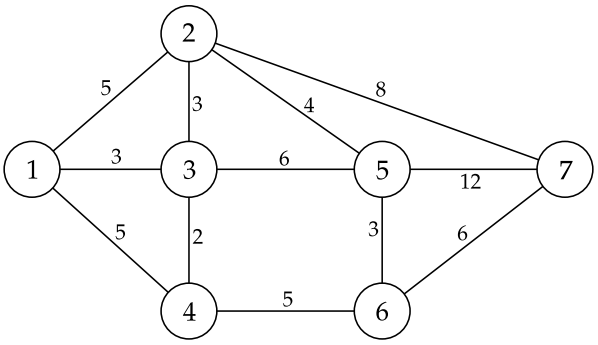
Rappel : En théorie des graphes, un arbre couvrant d'un graphe non orienté et connexe est un arbre qui est inclus dans ce graphe et qui connecte tous les sommets de ce graphe.

Soit $G = (V, E)$ un graphe connexe avec une valuation positive des arêtes. On cherche à construire T un arbre couvrant de poids minimum avec F, l'ensemble des arêtes de l'arbre.

Algorithm 1 Prim

Initialiser F à vide ;
Marquer arbitrairement un sommet ;
tant que *Il existe un sommet non marqué adjacent à un sommet marqué* **faire**
    Sélectionner un sommet y non marqué adjacent à un sommet marqué x tel que (x, y) est l'arête sortante de plus faible poids $F \coloneqq F(x, y)$;
    Marquer y ;
fin
F constitue un arbre couvrant de poids minimum ;

Écrire la fonction `prim` qui, à partir d'un graphe non orienté, renvoie les arêtes d'un arbre couvrant de poids minimal. Vous utiliserez la matrice d'adjacence associée au graphe suivant en considérant le noeud 1 comme noeud de départ :



Maintenant les arbres

Exercice 1

Définir une classe `Arbre(`*nom*`)` prenant en paramètre *nom* qui sera le nom de la racine de l'arbre. Ajouter les attributs `fg` et `fd` qui seront les sous-arbres gauche et droit, pour l'instant vide.

Exercice 2

Dans la classe `Arbre`, définir les méthodes suivantes :

- `insere_fg(Arbre)` qui renseigne le sous-arbre gauche de l'arbre et prenant en entrée un `Arbre` ;
- `insere_fd(Arbre)` qui renseigne le sous-arbre droit de l'arbre et prenant en entrée un `Arbre` ;

Exercice 3

Définir les deux arbres suivants, avec la classe `Arbre`.

Exercice 4

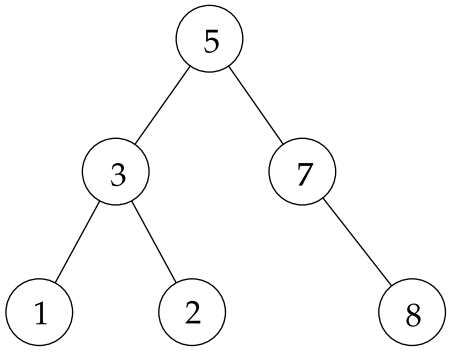
Dans la classe `Arbre`, définir une méthode `h` qui renvoie la hauteur de l'arbre.

Exercice 5

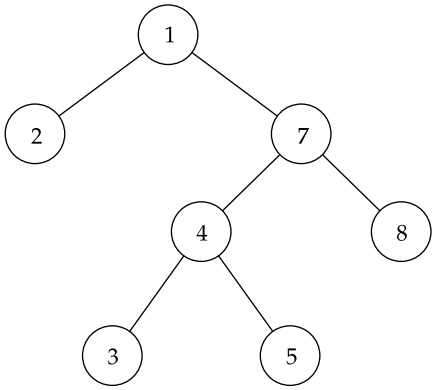
Rappel : Les parcours

- Préfixe : on affiche le noeud visité ; on visite le fils gauche s'il existe ; on visite le fils droit s'il existe.
- Infixe : on visite le fils gauche s'il existe ; on affiche le noeud visité ; on visite le fils droit s'il existe.
- Suffixe : on visite le fils gauche s'il existe ; on visite le fils droit s'il existe ; on affiche le noeud visité.

Dans la classe `Arbre`, implémenter les trois parcours (`prefixe`, `infixe`, `suffixe`). Dans cet exercice, on suppose que l'on commence les parcours depuis la racine. L'implémentation récursive est à privilégier.



Arbre$_1$



Arbre$_2$

Exercice 6

Dans la classe `Arbre`, définir la méthode `isBST(`*self*`)` qui renvoie `True` si l'arbre est un arbre binaire de recherche, sinon elle renvoie `False`.

Rappel : Une arbre est un arbre binaire de recherche lorsque chaque noeud possède une clé telle que chaque noeud du sous-arbre gauche ait une clé inférieure à celle du noeud considéré, et que chaque noeud du sous-arbre droit ait une clé supérieure à celle du noeud considéré.

Vous testerez la fonction sur les deux arbres de l'énoncé.

Exercice 7

Dans la classe `Arbre`, définir la méthode `isComplete(`*self*`)` qui renvoie `True` si l'arbre est un arbre binaire localement complet, sinon elle renvoie `False`.

Rappel : Une arbre binaire est localement complet si et seulement si tous ses noeuds ont 0 ou 2 fils.

Vous testerez la fonction sur les deux arbres de l'énoncé.

Exercice 8

Écrire une fonction `recherche_cleBST(Arbre,` *nom*`)` qui renvoie `True` si la clé *nom* est dans l'arbre binaire de recherche `Arbre`. Sinon il renvoie `False`. Tester cette fonction sur l'Arbre$_2$.