page 1 seule d'abord

1 Problématique

Pour faire travailler ses élèves sur la carte de course d'orientation le professeur d'EPS organise plusieurs petits jeux. Il réalise par exemple plusieurs parcours d'une balise à une autre. Cependant selon les contraintes météorologiques certains chemins peuvent ne plus être praticables.

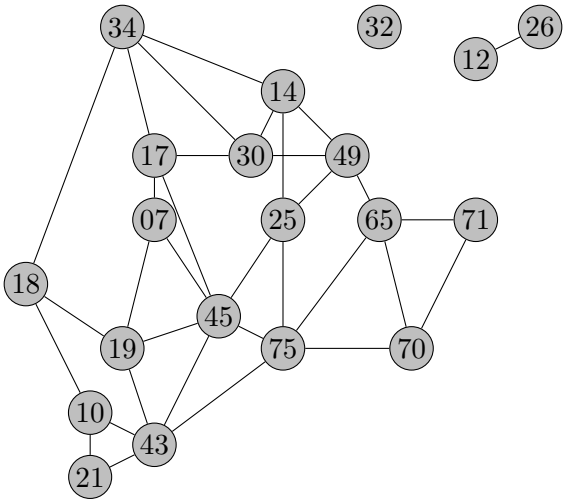


FIGURE 1 – Exemples de balises isolés

Peut-on construire des outils pour créer des parcours automatiquement et vérifier si toutes les balises sont atteignables ?

2 Connexité

2.1 Définition

Une **chaîne** est une liste ordonnée de sommets du graphe dans laquelle chaque sommet est adjacent au suivant. Un graphe est **connexe** quand deux sommets quelconque peuvent être reliés par une chaîne. Le graphe figure 1 n'est pas connexe car la balise 32 n'est pas atteignable depuis un autre sommet.

2.2 Parcours en profondeur (Depth First Search)

2.2.1 Méthodologie

Pour vérifier la connexité d'un graphe une stratégie possible est celle que nous pourrions appliquer dans un labyrinthe : depuis notre point de départ nous suivons un chemin jusqu'à être bloqué puis nous faisons demi-tour jusqu'à la précédente intersection. Pour illustrer ce fonctionnement nous nous appuierons sur le graphe 2.

algo pas précis ici volontairement : les élèves doivent détailler leur protocole dans l'activité 1. Notamment attention au cycle (et donc parcours en boucle).

Christophe Viroulaud

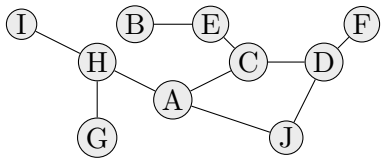


FIGURE 2 – Graphe étudié dans la suite

Activité 1 : Choisir un sommet et parcourir le graphe « à la main » en détaillant le protocole appliqué.

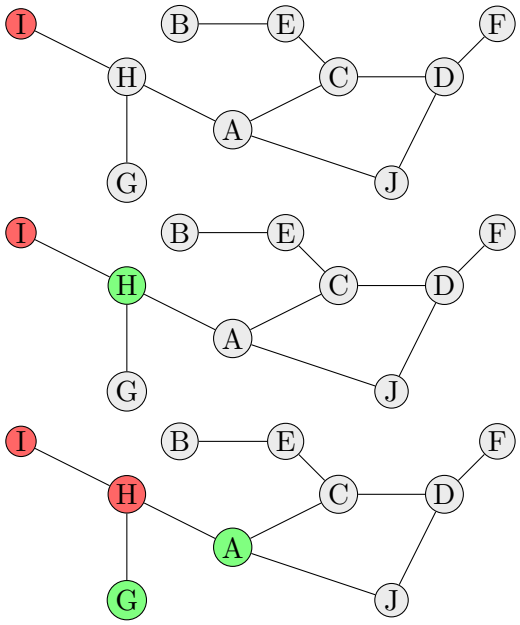
2.2.2 Formalisation

Détaillons l'algorithme du **parcours en profondeur**. Il est judicieux d'utiliser une pile pour stocker les sommets à visiter.

1. Initialisation : placer le sommet de départ dans la pile.
2. Tant qu'il reste des sommets dans la pile :
   (a) Dépiler le premier sommet de la pile.
   (b) S'il n'est pas marqué *visité* :
   — Le marquer *visité*.
   — Placer chaque voisin de ce sommet dans la pile.

Il faut remarquer que nous pourrions n'ajouter que les voisins non encore visités. Ceci ne change rien au niveau de l'efficacité car il faut tout de même vérifier à chaque fois si le sommet est dans *visités*.

Appliquons l'algorithme sur le graphe 2.



Initialisation :
le sommet I est empilé.
Sommets visités : ∅

Pile : | | | | I |

Le sommet I est dépilé et marqué *visité*.
Les voisins sont empilés.
Sommets visités : I

Pile : | | | | H |

Le sommet H est dépilé et marqué visité.
Les voisins sont empilés.
Sommets visités : I, H

Pile : | | A | I | G |

Activité 2 :

1. Dérouler l'algorithme « à la main » et vérifier sa correction.

Christophe Viroulaud

2. À l'aide de la classe *Graphe* du cours précédant, construire la représentation en mémoire du graphe 2.
3. Dans la classe *Graphe* ajouter la méthode **DFS(self, depart : str) → list** qui renvoie la liste des sommets visités avec un parcours en profondeur, en partant de *depart*.
4. Écrire la docstring de la méthode.

2.2.3 Efficacité

Chaque arête n'est examinée qu'une seule fois. La complexité de l'algorithme dépend donc directement du nombre d'arêtes du graphe.

2.3 Vérifier la connexité

Pour vérifier la connexité du graphe il suffit alors de vérifier si le nombre de sommets atteignables avec un parcours en profondeur correspond au nombre total de sommets du graphe.

Activité 3 : Écrire la méthode **est_connexe(self) → bool** qui renvoie *True* si le graphe est connexe.

3 Plus court chemin

3.1 Définition

Le plus court chemin entre deux sommets correspond au nombre minimal d'arêtes pour relier ces deux sommets. Il n'y a pas encore ici de notion de distance entre deux balises sur la carte de CO.

3.2 Parcours en largeur (Breadth First Search)

3.2.1 Méthodologie

La stratégie consiste en la visite de tous les sommets à la distance 1 du sommet de départ, puis tous ceux à la distance 2 et ainsi de suite. Pour chaque sommet de rang n nous pouvons accéder aux sommets de rang $n+1$. Donc en partant du sommet de distance 0, nous pouvons accéder à tous les autres, s'ils sont connexes.

Activité 4 : Choisir un sommet et parcourir le graphe « à la main » en détaillant le protocole appliqué.

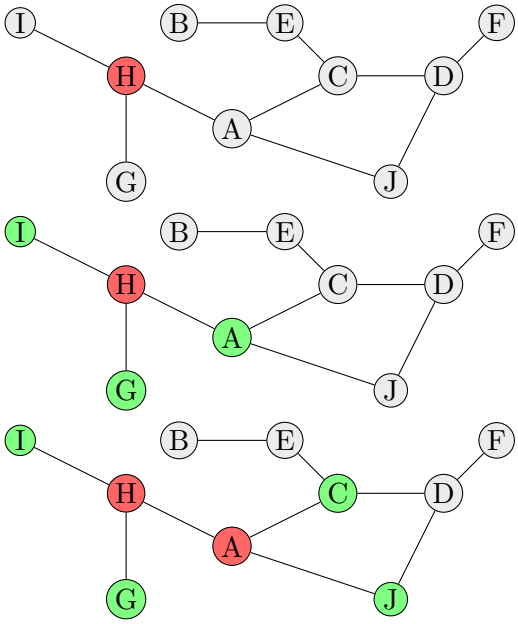
3.2.2 Formalisation

Détaillons l'algorithme du **parcours en largeur**. Il est strictement identique au parcours en profondeur mais cette fois il est nécessaire d'utiliser une file pour stocker les sommets à visiter.

1. Initialisation : placer le sommet de départ dans la file.
2. Tant qu'il reste des sommets dans la file :
   (a) Défiler le premier sommet de la file.
   (b) S'il n'est pas marqué *visité* :
   - Le marquer *visité*.
   - Placer chaque voisin de ce sommet dans la file.

Christophe Viroulaud

Appliquons l'algorithme sur le graphe 2.



Initialisation :
le nœud H est enfilé
File : | H | | | | | |
Nœuds visités : ∅

Le nœud H est défilé et marqué visité.
Les voisins sont enfilés.
File : | A | G | I | | | |
Nœuds visités : H

Le nœud A est défilé et marqué visité.
Les voisins sont enfilés.
File : | G | I | C | J | H | |
Nœuds visités : H, A

Activité 5 :

1. Dérouler l'algorithme « à la main » et vérifier sa correction.
2. Dans la classe *Graphe* ajouter la méthode **BFS(self, depart : str) → list** qui renvoie la liste des sommets visités avec un parcours en largeur, en partant de *depart*.

3.2.3 Efficacité

Dans ce cas encore l'algorithme est très efficace car chaque arête n'est examinée qu'une seule fois. La complexité dépend du nombre d'arêtes du graphe.

3.3 Trouver le plus court chemin

Pour trouver le chemin le plus court entre deux sommets il faut adapter le parcours en largeur pour qu'il retienne la distance entre *depart* et les autres sommets.

Activité 6 : Écrire la méthode **plus_court_chemin(self, depart : str, arrivee : str) → int**, adaptation de *BFS* et qui renvoie le chemin entre *depart* et *arrivee*. Il sera nécessaire d'utiliser un dictionnaire **distances = {depart : 0}** qui retiendra les distances entre les sommets et *depart*.

BFS récursif est plus compliqué (moins naturel que DFS)

4 Application au parcours de CO

Activité 7 :

1. Récupérer et modifier le fichier *json* pour réaliser la situation de la figure 1.
2. Tester les méthodes précédemment réalisées.
3. Écrire la méthode **plus_court_chemin_detail(self, depart : str, arrivee : str) → list**, adaptation de *plus_court_chemin* qui renvoie le chemin à parcourir pour atteindre *arrivee* depuis *depart*. Le processus se déroulera en deux étapes :

Christophe Viroulaud

- déterminer le prédécesseur de chaque sommet,
- puis reconstruire le chemin à partir des prédécesseurs. La méthode renverra *None* si la balise n'est pas atteignable.

Il est à noter que la méthode renvoie un chemin possible mais qu'il peut en exister d'autres.

Christophe Viroulaud