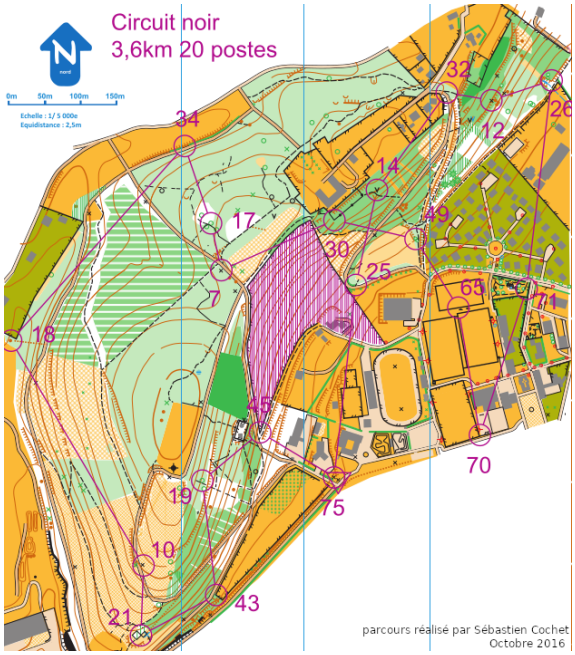
mettre **carte_co.zip** sur site ; donner d'abord les pages 1, 2 seulement

1 Problématique

La *course d'orientation* est une activité proposée par l'association sportive du lycée. C'est un sport très complet et apprécié des élèves. Cependant un inconvénient pour les enseignants qui organisent une séance est le temps de préparation nécessaire. Un support numérique peut permettre d'optimiser ce temps.



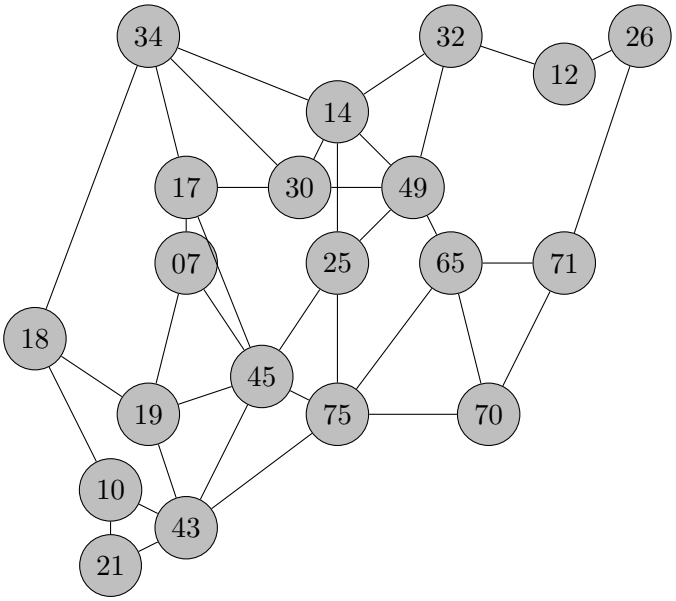


FIGURE 1 – Parcours noir

Comment peut-on représenter les balises de CO en mémoire ?

2 Notion de graphe

Un graphe est l'ensemble des **sommets** et des **arêtes** qui relient certains sommets entre eux. L'**ordre** du graphe est le nombre de sommets. Le **degré d'un sommet** est le nombre d'arêtes de ce sommet. Deux sommets reliés entre eux sont dits **adjacents**. Enfin un graphe est dit **complet** si tous les sommets sont adjacents.

Activité 1 :

1. Donner l'ordre et le degré du graphe figure 1.
2. Donner deux sommets adjacents.
3. Donner le degré du sommet 30.
4. Ce graphe est-il complet ?
5. Schématiser un graphe complet d'ordre 4.

Christophe Viroulaud

3 Représentations en mémoire

3.1 Matrice d'adjacence

On appelle **matrice d'adjacence** la représentation mathématique dont le terme a_{ij} vaut 1 si les sommets i et j sont reliés par une arête et 0 sinon.

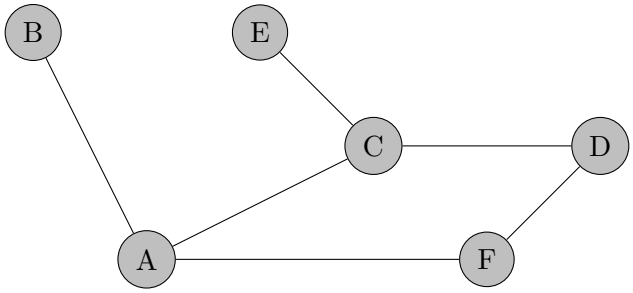


FIGURE 2 – Graphe étudié

La matrice d'adjacence du graphe 2 est :

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Activité 2 : Déterminer une structure de données permettant de représenter la matrice d'adjacence en mémoire.

On notera la symétrie dans la matrice. Cette observation s'explique aisément : L'arête de A vers B est également représentée par celle de B vers A.
Cette représentation est relativement simple à utiliser mais n'est pas toujours économe en ressource mémoire, particulièrement si le nombre d'arêtes est faible : la matrice sera *creuse*.

3.2 Dictionnaire d'adjacence

Une autre terminologie consiste à, pour chaque sommet, énumérer les sommets adjacents : c'est le **dictionnaire d'adjacence**.

- A : B, C, F
- B : A
- C : A, D, E
- D : C, F
- E : C
- F : A, D

Activité 3 : Construire le dictionnaire d'adjacence en mémoire.

littérature : existe aussi liste d'adjacence, liste successeur...
pour stocker les nœuds successeurs on peut utiliser une liste ou mieux un set : collection non triée d'objets distincts

Christophe Viroulaud

Cette représentation aura notre préférence. Contrairement à la matrice d'adjacence, il n'est pas nécessaire de connaître touts les sommets à l'avance pour la construire. De plus, à part pour les cas où pratiquement tous les sommets sont reliés entre eux, elle occupe moins d'espace.

3.3 Représentation de la carte de CO

Appliquons ces représentations à la carte de CO.

Activité 4 :

1. Représenter en mémoire la matrice d'adjacence et le dictionnaire d'adjacence de la carte (figure 1).
2. Récupérer le dossier compressé **carte_co.zip** sur le site https://cviroulaud.github.io
3. Importer la bibliothèque **mod_verification** et utiliser la fonction **verifier(graphe)**.

4 Utilisation de la programmation objet

Les graphes trouvent de multiples applications dans de nombreux domaines. Il ne paraît alors pas inutile de créer une classe générique que nous pourrons utiliser dans des situations différentes.
Nous préférerons l'utilisation d'un dictionnaire d'adjacence pour contenir les sommets.

Activité 5 :

1. Créer une classe **Graphe**. Elle possédera un attribut **sommets**, dictionnaire vide.
2. Créer la méthode **ajouter_sommet(self, s) → None** qui crée la clé *s* dans *sommets*. La valeur correspondante sera un *set* vide. Cette méthode vérifiera si le sommet n'est pas déjà présent avant de l'ajouter.
   Il faut noter que le paramètre *s* est un entier dans le cas de notre carte de CO mais nous pourrions avoir une chaîne de caractère ou même d'autres structures plus complexes.
3. Créer la méthode **ajouter_arete(self, s1, s2) → None** qui :
   — ajoute les sommets s'ils ne sont pas déjà présents,
   — crée les arêtes entre les deux sommets.
4. Créer la méthode **sont_relies(self, s1, s2) → bool** qui renvoie *True* s'il existe une arête entre s1 et s2.
5. Créer la méthode **get_adjacents(self, s) → set** qui renvoie les sommets voisins de *s*.
6. Créer la méthode **get_sommets(self) → list** qui renvoie la liste des sommets.
7. Créer une instance **parcours_noir** de la classe Graphe.
   Le fichier *parcours_noir.json* contient la structure du parcours. Ce type de fichier permet de stocker des données facilement. Python gère ce format avec la bibliothèque *json*. Les accolades sont transformées en dictionnaire et les crochets en liste.
8. Observer la structure des données dans le fichier *json*.
9. Importer le fichier en lecture dans le programme Python et remplir l'instance *parcours_noir* du *Graphe*. La méthode *load* de la bibliothèque *json* est suffisante.

pour les clés, il faut une structure non mutable (donc hashable)
on utilisera set() plutôt que list() car pour des grands graphes, set sera plus efficace
list(dictionnaire) renvoie la liste des clés

Christophe Viroulaud