Bac Général Maths + NSI (Mathématiques et Numérique et sciences informatiques)

SUJET D'EXAMEN (SESSION OFFICIELLE)

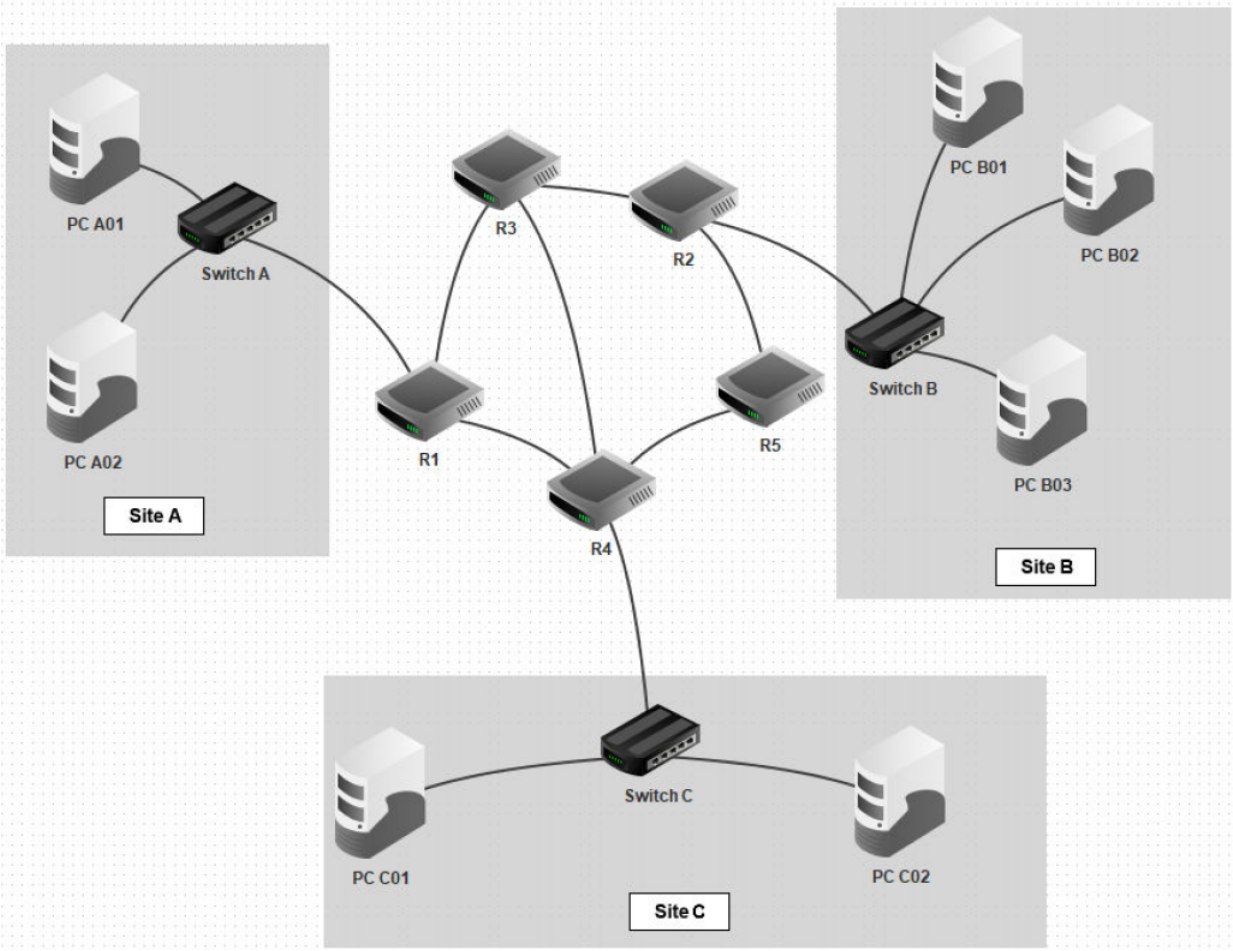
Numérique et sciences informatiques - Session 2026

Sujet de l'examen national tel que publié pour la session indiquée. Source : sujets nationaux publiés par le ministère chargé de l'examen et les services académiques. Document mis à disposition par FormaV pour la révision.

Toutes les annales et leurs corrigés : https://bac-maths-nsi.fr/annales-sujets-et-corriges

Métropole - juin 2026 - sujet 2

Exercice 1 (Réseaux, routage et POO - 6 points)
Une entreprise dispose d'une infrastructure réseau répartie sur plusieurs sites interconnectés à l'aide de routeurs. La figure ci-dessus représente le schéma de ce réseau, où les routeurs sont notés de R1 à R5.



Afin d'assurer la communication entre les différents postes et sous-réseaux, des protocoles de routage tels que RIP ou OSPF sont utilisés.

- Le protocole RIP minimise le nombre de sauts entre deux routeurs ; un « saut » correspond au transfert des données d'un routeur à un autre.
- Le protocole OSPF (*Open Shortest Path First*) minimise la somme des coûts des liaisons entre les routeurs empruntées par le paquet ; le coût entre deux routeurs voisins dépend du débit de la liaison entre ces routeurs selon la formule

$$\text{coût} = \frac{10^8}{\text{débit}},$$

où le débit est exprimé en bit/s et le coût est sans unité.

Le bon fonctionnement de ce réseau repose sur une configuration correcte des adresses IP, des masques de sous-réseau, des routes, ainsi qu'une gestion rigoureuse des ressources (comme la mémoire et les tables de routage).

Le réseau utilise des adresses IPv4, c'est-à-dire des adresses de la forme `IP/S` où :

- `IP` est une suite de quatre entiers entre 0 et 255 séparés par un point ;
- `S` est un entier entre 1 et 32.

L'adresse IP est codée en machine sur 4 octets, soit 32 bits. L'entier `S` indique que les `S` premiers bits de `IP` correspondent à la partie fixe de l'adresse et les suivants à la partie propre à la machine.

Deux adresses IP sont réservées :

- l'adresse du réseau local qui est constituée de la partie fixe de l'adresse complétée par des bits égaux à 0 ;
- l'adresse de diffusion (*broadcast*) qui est constituée de la partie fixe de l'adresse complétée par des bits égaux à 1.

On considère le poste PC C01 du site C ayant pour adresse IPv4 : `172.16.2.1/16`

1. Déterminer l'adresse IP du réseau local dédié au site C.
2. Déterminer l'adresse IP de diffusion du réseau local dédié au site C.
3. Donner le nombre maximal de machines que l'on peut connecter sur le réseau local dédié au site C, y compris les machines déjà présentes.
4. Recopier et compléter la table de routage du routeur R1 obtenue avec le protocole RIP.

Table de routage R1		
Destination	Passe par	Nombre de sauts
R2		
R3		
R4		
R5		

5. Déterminer le chemin que suit un paquet envoyé depuis PC A01 (Site A, relié à R1) vers PC B01 (Site B, relié à R2) en supposant qu'on utilise le protocole RIP.
6. Déterminer la nouvelle route qu'un paquet peut suivre de R1 à R2 si le routeur R3 tombe en panne, toujours en supposant qu'on utilise le protocole RIP.
7. Recopier et compléter le tableau suivant donnant le coût pour le protocole OSPF des liaisons réseau selon leur type de connexion.

Type de connexion	Débit (bit/s)	Coût
Ethernet	10^7	
Fast Ethernet	10^8	
Fibre	10^9	

Le tableau ci-dessous indique le type connexion pour chaque liaison du réseau.

Liaison	Type de connexion
R1-R3	Fibre
R1-R4	Ethernet
R2-R3	Ethernet
R5-R4	Fast Ethernet
R3-R4	Fast Ethernet
R5-R2	Fast Ethernet

8. Déterminer, en justifiant, le chemin choisi par le protocole OSPF entre R1 et R4.

On décide pour la suite de représenter en Python chaque routeur par une chaîne de caractères, par exemple `'R1'`, et de représenter les routes par des listes de routeurs, par exemple `['R1', 'R3', 'R2']`.

Un code Python permettant de définir une liste de routes (chaque route étant une liste de routeurs) a été créé. Ce code est composé :

- d'une liste vide appelée `liste_routes` qui permettra de contenir les routes ;
- d'une constante `MAX_ROUTES` indiquant le nombre maximum de routes que peut contenir la liste `liste_routes` ;
- d'une fonction `ajouter_route` qui ajoute la route donnée en paramètre à la liste `liste_routes`.

```
1 liste_routes = []
2 MAX_ROUTES = 5
3
4 def ajouter_route (route):
5     liste_routes.append(route)
```

9. Justifier que le code proposé ci-dessus ne permet pas de s'assurer que la contrainte d'un nombre maximal de routes contenues dans la liste est respectée.

On souhaite créer une classe `Routage` qui contient :

- le constructeur `__init__` qui permet d'initialiser un objet avec une capacité maximale donnée ;
- une méthode `ajouter` qui ajoute une route uniquement si la capacité maximale n'est pas atteinte ; sinon, elle affiche un message d'erreur.

Un objet de type `Routage` doit contenir les attributs :

- `capacite` : un nombre entier initialisé à 5 par défaut ;
- `routes` : une liste pour stocker les routes.

```
1  class Routage:
2      def __init__ (self, capacite = 5):
3          self.capacite = ...
4          self.routes = []
5
6      def ajouter (self, route):
7          if ... :
8              ...
9          else:
10             print("Erreur : capacité maximale atteinte")
```

10. Recopier et compléter les lignes 3, 7, 8 et 10 du code ci-dessus pour respecter les contraintes de la classe.
11. Écrire une méthode `afficher` de la classe `Routage` qui affiche toutes les routes présentes dans la liste `routes`. Par exemple si l'attribut `routes` de l'objet contient les routes `['R1','R3', 'R2']` et `['R1','R4', 'R5', 'R2']`, l'affichage sera :

```
R1
R3
R2
---
R1
R4
R5
R2
---
```

Exercice 2 (Mise au point de programme, gestion de bugs et graphes - 6 points)

Le *taquin* est un jeu qui se joue avec 15 tuiles numérotées dans une grille de 4 lignes et 4 colonnes pouvant en contenir 16. La tuile manquante permet de déplacer les tuiles adjacentes en les faisant glisser. Par exemple dans la grille de gauche de la figure 1, il est possible de déplacer les tuiles 12 et 15, et dans celle de droite, il est possible de déplacer les tuiles 14, 15 ou 3.

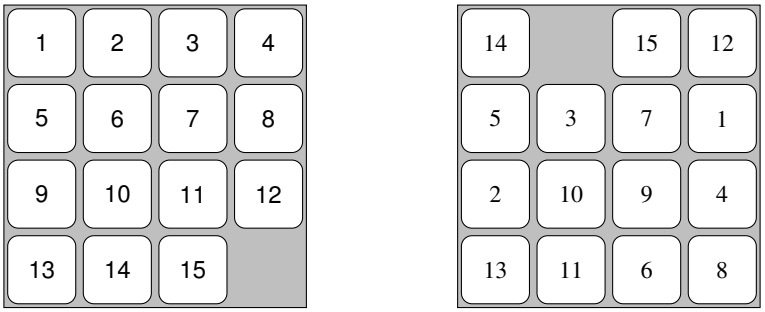


FIGURE 1 – Une grille rangée, à gauche, et mélangée, à droite

Le but du jeu, partant d'une grille mélangée, est de ranger la grille en procédant à des glissements successifs afin de remettre les tuiles comme sur la grille de gauche sur la figure 1. Une grille mélangée pour laquelle cela est possible est dite *résoluble*.

On souhaite, dans cet exercice, écrire des fonctions permettant de mélanger une grille de taquin tout en s'assurant que la grille mélangée est résoluble.

Partie A : mélange aléatoire

Une grille de taquin est représentée en machine par une liste Python contenant quatre sous-listes de quatre entiers. Chacune des sous-listes contient des valeurs entières de façon à ce que chaque entier entre 1 et 16 apparaisse une unique fois. La valeur 16 représente la tuile vide. La grille rangée ci-dessus est ainsi représentée par :

```
rangee = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]]
```

On suppose que la liste `melangee` représente la grille mélangée de droite sur la figure 1.

1. Donner la valeur de `melangee[2][1]`.

On décide dans un premier temps d'obtenir une grille mélangée en utilisant la démarche suivante :

- on crée une liste `valeurs` contenant les entiers entre 1 et 16. Dans la suite du sujet, cette liste sera appelée *liste aplatie* ;
- on mélange la liste aplatie en utilisant la fonction `random.shuffle` de Python ;
- on convertit la liste aplatie ainsi mélangée en une liste de listes afin d'obtenir la *grille*.

Afin de créer la liste aplatie `valeurs` on saisit tout d'abord l'instruction `valeurs = [k for k in range (16)]`. On constate que les valeurs ne sont pas celles attendues.

2. Réécrire cette instruction en la corrigeant pour que la liste `valeurs` contienne bien les entiers de 1 à 16.

La liste aplatie `valeurs` étant désormais correctement créée, on la mélange à l'aide de la fonction `random.shuffle`. La documentation de Python indique que `random.shuffle(x)`, mélange la séquence `x` sans créer de nouvelle instance (« en place »). On saisit donc :

```
>>> from random import shuffle
>>> valeurs = shuffle(valeurs)
>>> print(valeurs)
None
```

On constate que la variable `valeurs` est désormais affectée à `None`.

3. Proposer une modification de la ligne `valeurs = shuffle(valeurs)` afin de corriger cette erreur.

La fonction `en_grille` donnée ci-dessous prend en paramètres une liste d'entiers `valeurs` et un entier `n`. Cette fonction permet de transformer la liste `valeurs` (contenant `n` × `n` éléments) en une liste de `n` listes contenant chacune `n` entiers.

```
1 def en_grille(valeurs, n):
2     assert ...
3     grille = [[0 for j in range(n)] for i in range(n)]
4     for i in range(n):
5         for j in range(n):
6             grille[i][j] = valeurs[j * n + i]
7     return grille
```

4. Recopier et compléter la ligne 2 de cette fonction afin qu'elle vérifie que la liste passée en paramètre contient le bon nombre d'éléments.

L'appel `en_grille([1, 2, 3, 4], 2)` renvoie `[[1, 3], [2, 4]]` au lieu de `[[1, 2], [3, 4]]`.

5. Recopier et modifier la ligne 6 de la fonction `en_grille` afin de corriger cette erreur.

Partie B : grille résoluble

Certaines grilles obtenues par la méthode précédente ne sont pas résolubles. Pour savoir si une grille est résoluble, on doit calculer :

- le **nombre d'inversions** présentes dans la liste aplatie ;
- la **distance**, mesurée en nombre de déplacements, séparant la tuile vide de sa position finale en bas à droite de la grille.

Une **inversion** dans une liste aplatie `valeurs` est un couple d'indices `(i, j)` vérifiant `i < j` et `valeurs [i] > valeurs [j]`. Par exemple, la liste `valeurs = [6, 9, 7, 8]` compte deux inversions : les couples `(1, 2)` (car $9 > 7$) et `(1, 3)` (car $9 > 8$).

La **distance** séparant la tuile vide de sa position finale est égale à la somme du nombre de lignes et du nombre de colonnes séparant la tuile vide et le coin inférieur droit de la grille. Dans la grille de droite de la figure 1 cette distance vaut 5. En effet, la tuile vide est séparée de 3 lignes et de 2 colonnes de sa position finale.

On admet que la grille est résoluble si et seulement si la somme du nombre d'inversions et de la distance est un nombre pair.

6. Indiquer si la grille représentée en machine par la liste suivante est résoluble ? Justifier.

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 16, 15, 14]]
```

On propose ci-dessous la fonction `compte_inversions` qui permet de compter les inversions dans une liste aplatie passée en paramètre.

```
1 def compte_inversions(valeurs):
2     total = 0
3     for i in range(len (valeurs)):
4         for j in range(len (valeurs)):
5             if valeurs[i] > valeurs[j]:
6                 total = total + 1
7     return total
```

On remarque que la grille non mélangée de 2 lignes et 2 colonnes compte 0 inversion, ainsi on peut tester cette fonction, en exécutant le test suivant :

```
assert compte_inversions ([1, 2, 3, 4]) == 0
```

7. Proposer un test faisant intervenir une liste aplatie de quatre éléments et comptant deux inversions permettant de vérifier que la fonction `compte_inversions` renvoie le résultat attendu.
8. La fonction proposée ne passe pas les tests. Proposer une correction de la fonction `compte_inversions`.

On propose désormais la fonction `distance_tuile_vide` qui prend en paramètre une liste de listes et un entier `n`, où la liste représente une grille de taquin de `n` lignes et `n` colonnes, et qui renvoie la distance séparant la tuile vide de sa position finale.

```
1 def distance_tuile_vide(grille, n):
2     num_tuile_vide = n * n
3     for i in range(n):
4         for j in range(n):
5             if grille[...][...] == ...:
6                 return ...
```

9. Recopier et compléter les lignes 5 et 6 de cette fonction afin qu'elle renvoie la distance attendue.

La fonction `est_resoluble` prend en paramètre la liste aplatie `valeurs` et un entier `n`, où la liste `valeurs` a `n×n` éléments, et renvoie le booléen indiquant si la grille de taquin qu'elle représente est résoluble ou non.

```
1 def est_resoluble(valeurs, n):
2     grille = en_grille(valeurs, n)
3     inv = ...(...)
4     dis = ...(...)
5     return (... + ...) ... == ...
```

10. Recopier et compléter cette fonction afin qu'elle renvoie la valeur attendue. On rappelle que `7%2` est égal à `1`, alors que `8%2` est égal à `0`.

Partie C : mélange réaliste

On souhaite désormais utiliser une méthode de mélange garantissant que la grille obtenue est résoluble. Pour ce faire, on va effectuer des déplacements réalistes en échangeant, à plusieurs reprises, la tuile vide avec une de ses tuiles voisines. On effectue ces échanges directement sur **la liste de valeurs aplatie**.

Pour représenter les déplacements réalisables, on définit un graphe dans lequel les sommets sont les indices des cases dans la liste aplatie. Deux sommets `i` et `j` sont reliés par une arête s'il est possible, lorsque la tuile vide est sur l'indice `i`, de l'échanger avec la tuile d'indice `j`. On représente ce graphe en machine par une liste d'adjacence `graphe` sous la forme d'une liste Python dans laquelle la valeur à l'indice `i` contient la liste des indices des cases avec lesquelles on peut échanger la tuile d'indice `i`.

On fournit ci-dessous, **à titre d'exemple**, le graphe des déplacements réalisables pour une grille de taquin de 3 lignes et 3 colonnes (soit 9 tuiles au total).

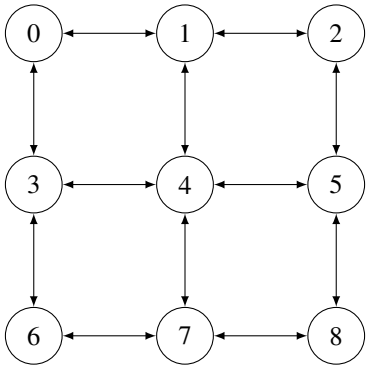


FIGURE 2 – Le graphe des déplacements pour un taquin à 3 lignes et 3 colonnes

On suppose que `graphe_3_3` représente ce graphe par sa liste d'adjacence. Ainsi `graphe_3_3[8]` vaut `[5, 7]` ou `[7, 5]`.

11. Donner une valeur possible de `graphe_4_4[9]` en supposant que `graphe_4_4` est la liste d'adjacence représentant le graphe associé à une grille à 4 lignes et 4 colonnes.

La fonction `melange_graphe` prend en paramètre un nombre entier `nb_dep`, un entier `n` et une liste d'adjacence `graphe` représentant un graphe de déplacements sur un taquin à `n` lignes et `n` colonnes et effectue `nb_dep` déplacements voisins successifs de la tuile vide. Ces déplacements sont choisis aléatoirement parmi ceux enregistrés dans le graphe grâce à la fonction `random.choice` qui renvoie un élément pioché aléatoirement dans la liste passée en paramètre.

```
1  from random import choice
2
3  def melange_graphe(nb_dep, n, graphe):
4      valeurs = [i+1 for i in range(n*n)] # liste aplatie rangée
5      num_tuile_vide = n * n
6      actuelle = num_tuile_vide - 1 # position de la tuile vide
7      for k in range(...):
8          prochaine = choice(...)
9          valeurs[actuelle] = valeurs[...]
10         valeurs[prochaine] = ...
11         actuelle = prochaine
12     return en_grille(valeurs, n)
```

12. Recopier et compléter les lignes 7 à 10 de la fonction `melange_graphe`.

Exercice 3 (Bases de données, files et graphe - 8 points)
Partie A
Dans cette partie, on pourra utiliser les clauses du langage SQL pour :

- construire des requêtes d'interrogation à l'aide de `SELECT`, `FROM`, `WHERE` (avec les opérateurs logiques `AND` et `OR`), `JOIN ... ON` ;
- construire des requêtes d'insertion et de mise à jour à l'aide de `UPDATE`, `INSERT` et `DELETE` ;
- affiner les recherches à l'aide de `DISTINCT` et `ORDER BY`.

Dans une entreprise qui possède plusieurs sites de production, on met en place une application de covoiturage pour aider les salariés à limiter leurs déplacements en voiture individuelle, et ainsi, diminuer l'empreinte carbone de l'entreprise. Avec cette application, chaque membre du personnel peut proposer un trajet ou s'inscrire sur un trajet proposé.

L'application repose sur une base de données avec trois tables, que l'on décrit par le schéma suivant où les attributs qui servent de clé primaire sont soulignés et les attributs qui servent de clé étrangère sont précédés du caractère `#`.

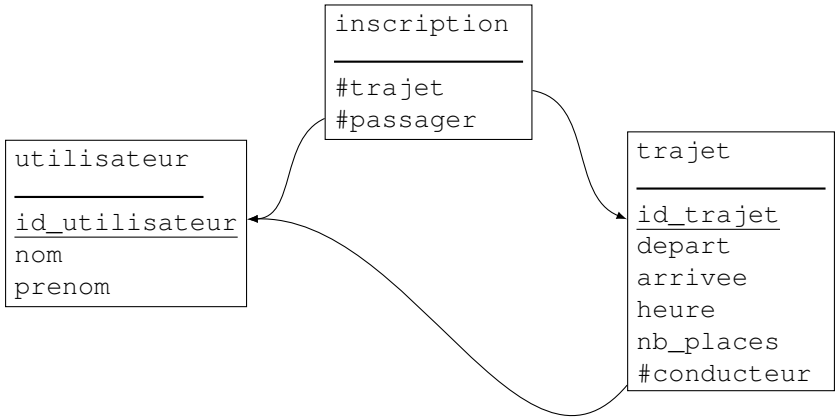


FIGURE 1 – Schéma de la base de données

1. Expliquer pourquoi les attributs `nom` et `prenom` n'ont pas été retenus comme clé primaire de la table `utilisateur`.

Les attributs `id_utilisateur` et `id_trajet` sont des nombres entiers.

2. Justifier que les attributs de la relation `inscription` sont aussi des nombres entiers.

La requête :

```
SELECT * FROM trajet
WHERE heure > '2026-06-19 00:00:00' AND heure < '2026-06-20 00:00:00'
```

renvoie le tableau suivant :

trajet					
id_trajet	depart	arrivee	heure	nb_places	conducteur
1291	Liverdun	Toul	2026-06-19 07:30:00	3	25
1292	Allain	Nancy	2026-06-19 07:45:00	2	30
1293	Messein	Nancy	2026-06-19 08:00:00	2	10
1294	Nancy	Messein	2026-06-19 18:20:00	2	10
1295	Nancy	Allain	2026-06-19 17:45:00	3	25
1296	Toul	Liverdun	2026-06-19 18:00:00	2	30

Avec des requêtes adaptées, on peut obtenir les informations suivantes concernant les inscriptions et les utilisateurs concernés par ces trajets :

inscription	
trajet	passager
1291	4
1291	15
1291	38
1292	18
1295	4
1295	15
1295	38
1296	18

utilisateur		
id_utilisateur	nom	prenom
4	Mizab	Yasmine
10	Di Maria	Alexis
15	Rey	Maxime
18	Nguema	Basil
25	Daniel	Valérie
30	Sanches	Nathalie
38	Fabre	Clément

3. Recopier et compléter la requête suivante afin de connaître le nombre de passagers inscrits sur le trajet n°1291.

```
SELECT COUNT(*) FROM ... WHERE ...
```

4. Écrire une requête permettant d'obtenir la liste des trajets qui arrivent à Nancy le 19 juin 2026 dans l'ordre croissant de l'heure de départ.

On suppose que le trajet n°1295 a été ajouté dans la base de données lors de sa création par une simple requête et qu'il n'a pas été modifié depuis.

5. Écrire une requête qui aurait permis d'ajouter le trajet n°1295 dans la base de données lors de sa création.
6. Écrire une requête permettant de modifier l'heure de départ à 18h40 pour le trajet n°1294. On pourra utiliser les valeurs présentes dans les tableaux extraits de la base de données.

Pour supprimer le trajet de 18h00 entre Toul et Liverdun, on essaye la requête suivante :

```
DELETE FROM trajet WHERE id_trajet=1296
```

Le gestionnaire de base de données donne alors l'erreur suivante :

```
ERROR 1451 (23000) at line 95 in file: 'covoit.sql': Cannot
delete or update a parent row: a foreign key constraint fails
```

7. Expliquer en détail les raisons de cette erreur.
8. Écrire une requête permettant d'obtenir la liste des noms et prénoms des passagers transportés au moins une fois par l'utilisatrice dont l'identifiant est 25.

Partie B

Alice, Maxime, Valérie et Clément doivent choisir un point de rendez-vous pour leur trajet de covoiturage. Ils ont identifié trois points de rendez-vous possibles. Dans le graphe suivant :

- les sommets A, M, C, et V représentent les domiciles des quatre covoitureurs ;
- les sommets P1, P2, et P3 représentent les points de rendez-vous possibles ;
- les arêtes représentent des trajets entre ces lieux qui possèdent tous approximativement les mêmes conditions de parcours (distance et durée).

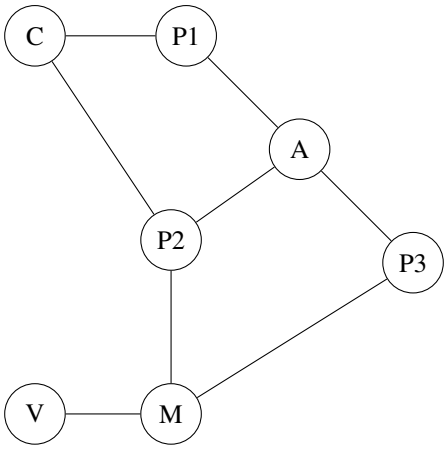


FIGURE 2 – Graphe des domiciles et points de rendez-vous

9. Écrire un dictionnaire Python représentant le graphe de la figure 2. Ce dictionnaire doit associer à chaque sommet la liste de ses sommets voisins.

On utilise la fonction `dico_distance` donnée ci-après pour déterminer toutes les distances en nombre d'arêtes entre un point de rendez-vous et les autres sommets du graphe.

```
1  def dico_distance(graphe, depart):
2      """ graphe : dictionnaire
3      {sommet : liste de sommets adjacents }
4      depart : un sommet du graphe
5      renvoie un dictionnaire
6      {sommet atteignable depuis depart :
7      distance entre depart et ce sommet }"""
8
9      # initialisation d'une file
10     file = File()
11     file.inserer(depart)
12
13     # initialisation du dictionnaire de sortie
14     dico = {depart : 0}
15     while not file.est_vide():
16         # extraire le prochain sommet à explorer
17         sommet = file.extraire()
18         # pour chaque voisin encore non atteint
19         for voisin in graphe[sommet]:
20             if voisin not in dico:
21                 # attribuer la distance à ce voisin
22                 dico[voisin] = dico[sommet] + 1
23                 # placer ce voisin dans la file
24                 # pour une exploration future
25                 file.inserer(voisin)
26     return dico
```

La fonction `dico_distance` utilise une file pour gérer l'ordre d'exploration des sommets du graphe. Cette file est implémentée avec une classe `File`. En tapant la commande `help(File)` dans la console de Python on obtient :

```
class File(builtins.object)
 |
 |  Methods defined here:
 |
 |  __init__(self)
 |     initialise une file vide
 |
 |  consulter(self)
 |     renvoie le premier élément disponible
 |     sans le retirer de la file
 |
 |     lève une exception IndexError si la file est vide
 |
 |  est_vide(self)
 |     renvoie True si la file est vide, False sinon
 |
 |  extraire(self)
 |     renvoie le premier élément disponible et
 |     le retire de la file
 |
 |     lève une exception IndexError si la file est vide
 |
 |  inserer(self, element)
 |     ajoute l'élément donné dans la file
```

10. Rappeler le principe de fonctionnement d'une file.
11. Donner un ordre possible d'insertion des sommets dans la file lorsqu'on applique la fonction `dico_distance` sur le graphe de la figure 2 avec le sommet P1 pour sommet de départ.
12. Donner le nom du type de parcours de graphe, utilisé dans la fonction `dico_distance`.

On définit l'*excentricité* d'un sommet dans un graphe comme la plus grande distance parmi les distances entre ce sommet et un autre sommet quelconque du graphe.

13. Recopier et compléter la fonction `excentricite` suivante qui calcule l'excentricité d'un sommet dans un graphe. L'utilisation de la fonction `max`, disponible sans importation, n'est pas autorisée.

```
1  def excentricite(graphe, sommet):
2      """ graphe : dictionnaire
3      {sommet : liste de sommets adjacents }
4      renvoie l'excentricité de sommet
5      dans le graphe (int) """
6      dico = dico_distance(graphe, sommet)
7      ...
8      ...
9      ...
10     ...
11     ...
```

14. Donner, en le justifiant avec un indicateur, le meilleur point de rendez-vous pour ces quatre personnes (Alice, Maxime, Valérie et Clément).