# Interrogation Écrite de Fin de Semestre ISN3
## 2ème année - Février 2025

**Durée totale :** *1h30*
**Documents autorisés :** *Synthèse personnelle de 2 pages format A4 (1 feuilles).*

- Le barème est sur 19 points, auxquels s'ajoutent : .
  - 1 point issu de l'évaluation par les pairs de l'APP
  - un modificateur lié aux parcours Pix : de -0,5 à +0,5 points
  - un modificateur lié à l'évaluation par les pairs : de -0,5 à +0,5 points
- Le sujet est sur 8 pages - il y a 3 exercices.
- **Rappel :** Toute fonction demandée dans une question à le droit d'être utilisée dans une autre question, même si elle n'a pas été écrite.

Nom : Prénom : Groupe :

## Exercice 1 : Questions de cours (3 pts)

*(1.1) Donnez la définition d'un arbre (0.25pt)*

graphe connexe sans cycle

*(1.2) Donnez la définition d'un arbre couvrant d'un graphe (0.25pt)*

arbre dont les sommets sont tous les sommets du graphe et les arêtes sont des arêtes du graphe

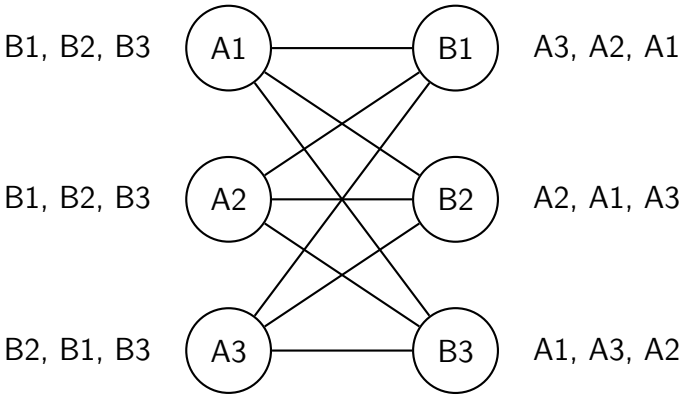
*(1.3) Donnez le nombre d'arêtes d'un arbre à $n$ sommets (0.25pt)*

$n-1$

*(1.4) Donnez le nombre d'arêtes d'un graphe non orienté complet à $n$ sommets. Justifiez (0.5pt)*

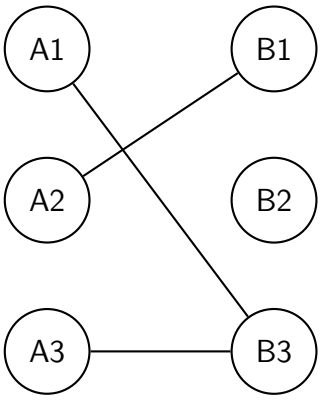
Graphe sans boucle : c'est l'ensemble de tous les sous-ensembles de 2 éléments différents $\binom{n}{2} = \frac{n(n-1)}{2}$. Autrement dit, chacun des $n$ sommets est adjacents aux $n-1$ autres et on divise $n(n-1)$ par 2 car non orienté.
Graphe avec boucle : c'est l'ensemble de tous les sous-ensembles de 2 éléments, pas nécessairement distincs $\frac{n(n-1)}{2} + n = \frac{n(n+1)}{2}$. Autrement dit, comme les graphes sans boucle, sauf qu'on rajoute $n$ boucles.

Voici un graphe biparti sur lequel on veut calculer un appariement avec l'algorithme de Gale-Shapley. Les préférences se lisent ainsi : $A1$ préfère être affecté à $B1$, puis à $B2$, puis à $B3$, tandis que $B2$ préfère être associé à $A2$, puis à $A1$, et sinon à $A3$.



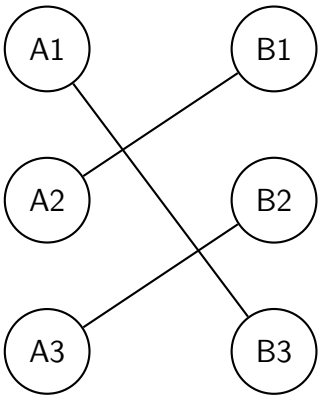

*(1.5) Pour chacun des 3 cas suivants* ***nommez la propriété*** *de l'algorithme de Gale-Shapley qui n'est pas respectée et expliquez brièvement pourquoi. (0.25 par cas)*

Cas 1



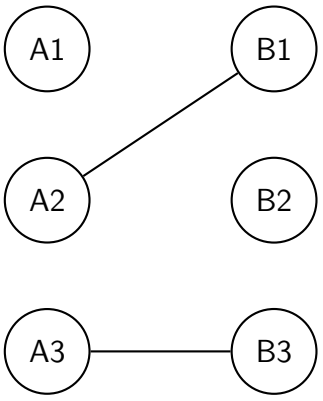

Pas un couplage : le degré de B3 est $> 1$

Cas 2




Couplage pas stable : A1 préfère B2 à B3 et B2 préfère A1 à A3

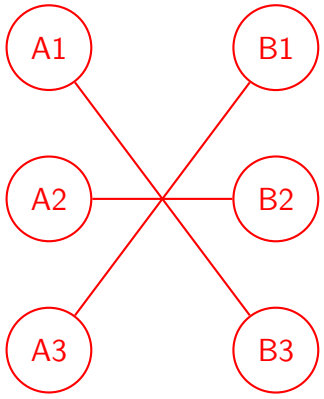
Cas 3




Couplage pas parfait : A1 et B2 non affectés

*(1.6) Dessinez l'affectation que calcule l'algorithme de Gale-Shapley (1pt)*

Résultat Gale Shapley



## Exercice 2 : Épidémiologie (9 pts)

On cherche à étudier la propagation d'une maladie au sein d'un groupe de personnes ayant des interactions plus ou moins nombreuses entre elles. Une personne malade va contaminer les personnes saines avec qui elle a des interactions. Plus il y a d'interactions, plus la contamination est rapide.

Un graphe modélise cette notion d'interactions : chaque sommet représente une personne, chaque arête représente l'existence d'interactions entre 2 personnes et la pondération représente la quantité des interactions entre ces 2 personnes. La figure 1 est un exemple d'interactions entre 6 personnes (A à F). Dans cet exemple, on pourra lire, par exemple, que la personne D a plus d'interactions avec C (14) qu'avec A (8), E (8) ou F (7), et n'en a pas avec B (pas d'arête).

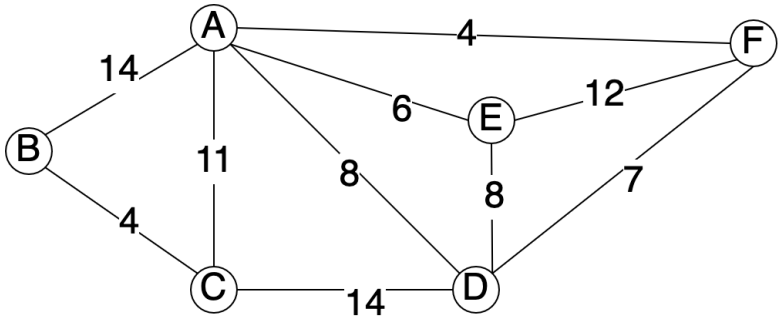



FIGURE 1 – Exemple de graphe d'interactions entre 6 personnes

On considère qu'un graphe pondéré non orienté connexe décrivant un tel système est stocké sous forme d'un dictionnaire d'adjacence dans la variable `graphe`.

Soit la fonction `nb_tours_epidemie` suivante :

```
1  def nb_tours_epidemie(graphe, depart):
2      malades = [depart]
3      nb_tours = 0
4      while len(malades)<len(graphe):
5          nb_tours+=1
6          contamines = []
7          for sommet in malades:
8              voisins = graphe.get(sommet,{})
9              v_max = None
10             poids_max = -math.inf
11             for v,p in voisins.items():
12                 if v not in malades and v not in contamines and p > poids_max:
13                     v_max = v
14                     poids_max = p
15             if v_max is not None:
16                 contamines.append(v_max)
17         malades = malades + contamines
18     return nb_tours
```

*(2.1) Déroulez ce programme sur le graphe de la figure 1, avec `depart = A`. Écrivez le contenu de la variable `malades` à chaque fin de tour de la boucle `while`. Quelle est la valeur `nb_tours` renvoyée à la fin du déroulé sur cet exemple ? (1 pt)*

États successifs de malades :
- [A] <— optionnel, car est avant le début de la boucle.
- [A, B]
- [A, B, C] – Si D ajouté à cette étape, alors erreur dans l'ordre de parcours
- [A, B, C, D] – Si E ajouté, alors erreur dans l'ordre de parcours
- [A, B, C, D, E, F] – Si E ou F pas ajouté, alors erreur dans le parcours des pondérations

nb_tours vaut 4

*(2.2) Expliquez ce que calcule cette fonction et ce que font ses différentes parties (ses différentes boucles)* ***en utilisant le vocabulaire du problème (personnes, interaction, malade, sain, contamination, etc.)*** *(1 pt)*

Dans cette fonction, une personne malade peut contaminer 1 de ses voisins sains à chaque tour. Cette fonction calcule le nombre de tours de contamination (ou le temps de propagation de la maladie) à partir d'une personne malade avant que tout le monde ne soit malade, sachant qu'à chaque tour, chaque personne malade ne peut contaminer qu'1 personne saine.
- while : tant que tout le monde n'est pas malade, on entame un nouveau tour
- for 1 : pour chaque personne malade, on tente de contaminer un voisin
- for 2 : on cherche le voisin sain avec lequel il y a eu le plus d'interaction, et on le contamine

*(2.3) Combien de fois la boucle* `while` *est exécutée dans le pire des cas (dans un graphe pondéré connexe non orienté à $n$ sommets) ? Justifiez. (0,5pt)*

À chaque étape, on passe en revue tous les voisins des sommets malades et on en contamine au moins 1. Donc $n-1$ itérations maximum (où $n$ le nombre de sommets).

*(2.4) Écrivez la fonction* `centre_epidemie` *prenant un graphe d'interactions en paramètre, et qui cherche (et renvoie) le sommet de départ minimisant le nombre d'étapes nécessaires pour contaminer tout le graphe. (1,5pt)*

```
1 def centre_epidemie(graphe):
2     tours_min = math.inf
3     sommet_min = None
4     for sommet in graphe:
5         tours = nb_tours_epidemie(graphe, sommet)
6         if tours < tours_min:
7             tours_min = tours
8             sommet_min = sommet
9     return sommet_min
```

**Vous n'avez pas besoin des réponses aux questions précédentes pour faire la suite.**

Au sein de la population, on connait une liste des personnes qui sont vaccinées et une liste de personnes qui sont fragiles. Nous faisons l'hypothèse que le vaccin est fiable à 100% (une personne vaccinée ne peut ni être malade, ni contaminer quelqu'un). Chaque fois que le centre de contrôle des épidémies est notifié de l'apparition d'une personne malade, il souhaite vérifier s'il y a un risque qu'une personne fragile soit contaminée.

**Zone de réponse pour les deux prochaines questions sur la page suivante.**.

(2.5) *Écrivez une fonction `danger_contamination` qui prend en paramètre un graphe des interactions, un sommet malade, une liste de sommets vaccinés et une liste de sommets fragiles, et qui renvoie un booléen valant `True` si au moins une personne fragile peut-être contaminée par propagation de cette maladie à partir de la personne malade (et renvoyant `False` sinon). Cette fonction pouvant être exécutée sur de très grands graphes, vous ferez attention à l'optimisation en temps d'exécution de votre solution. (2pts)*

```
1  def danger_contamination(graphe, start, vaccines, fragiles):
2      parents = {}
3      for u in vaccines:
4          parents[u] = None  # pour empêcher d'utiliser ces sommets
5
6      stops = {}
7      for s in fragiles :
8          stops[s] = None
9
10     vaccin_suffisant = True
11     a_traiter = [start]
12     while len(a_traiter) > 0 and vaccin_suffisant :
13         next = a_traiter.pop()
14         for voisin, poids in graphe[next].items():
15             if voisin not in parents:
16                 parents[voisin] = next
17                 a_traiter.append(voisin)
18
19             if voisin in stops :
20                 vaccin_suffisant = False
21
22     return not vaccin_suffisant
```

(2.6) *Listez les optimisations que vous avez faites dans votre code et expliquez pourquoi cela réduit le temps d'exécution sur un grand graphe. (3pts)*

Pour exemple, voici les optimisations faites sur la correction proposée :

- utiliser parents en non pas *a_traiter* pour le test de "déjà vu" (ligne 15), car c'est un dictionnaire : O(1) pour le test d'appartenance au lieu de O(n) si fait sur *a_traiter*.
- while principal s'arrête dès qu'une personne fragile atteinte.
- vaccines et fragiles placés dans des dictionnaires : cout O(n) pour faire le transfert, au lieu de O(n) pour chaque test d'appartenance. L'optimisation consiste à convertir les 2 listes en dictionnaire.
- pile et non pas file : pas de pop(0) qui serait en O(n) car déplacement de l'ensemble des éléments de la liste

Autres exemples d'idées d'optimisation valides :

- construction d'une copie du graphe en enlevant tous les sommets vaccinés
- ...

## Exercice 3 : État civil (7 pts)

Dans cet exercice, on appellera *État civil* d'une ville la mémoire de l'ensemble des personnes qui y sont nées. Ces informations sont stockées dans une structure de dictionnaires.

Chaque personne a un identifiant qui est la clé du dictionnaire et est décrite par les champs suivants : *nom*, *prenom*, *date_naissance*, *id_parent_1*, *id_parent_2*. Un *id_parent* qui vaut *None* signifie que le parent correspondant n'est pas connu.

Exemple de dictionnaire d'état civil :

```
{'1': {'nom': 'Curie', 'prenom': 'Marie', 'date_naissance': '07/11/1867',
       'id_parent_1': None, 'id_parent_2': None},

 '2': {'nom': 'Curie', 'prenom': 'Pierre', 'date_naissance': '15/05/1859',
       'id_parent_1': None, 'id_parent_2': None},

 '3': {'nom': 'Joliot-Curie', 'prenom': 'Irène', 'date_naissance': '12/09/1897',
       'id_parent_1': '1',  'id_parent_2': '2'},

 '4': {'nom': 'Joliot', 'prenom': 'Frédéric', 'date_naissance': '19/03/1900',
       'id_parent_1': '2', 'id_parent_2': None},

 '5': {'nom': 'Joliot', 'prenom': 'Pierre', 'date_naissance': '12/03/1932',
       'id_parent_1': '3',  'id_parent_2': '4'}
}
```

Dans ce format, l'état civil ne contient que les `id` des parents. On souhaite pouvoir compléter cet état civil avec la liste des enfants connus.

*(3.1) Écrivez une fonction `maj_enfants` qui prend en paramètre un état civil et qui met à jour la liste des enfants. Cette fonction ajoute **ou complète** la liste des id des enfants de chaque personne dans l'état civil. Si besoin, cette fonction ajoute une nouvelle clef dans le dictionnaire de chaque personne ayant des enfants (2 pts)*

```python
def maj_enfants(etat_civil):
    # Pour chaque "enfant" id
    for id, personne in etat_civil.items() :
        # Si id a un parent 1
        id_p1 = personne['id_parent_1']
        if id_p1 != None :
            # Ajouter id en tant qu'enfant auprès de son parent 1 si nécessaire
            p1 = etat_civil[id_p1]
            l = p1.get("ids_enfants",[])
            # Enfant déjà présent ?
            if id not in l :
                l.append(id)
                p1['ids_enfants'] = l


        # Idem si id a un parent 2
        id_p2 = personne['id_parent_2']
        if id_p2 != None :
            l = etat_civil[id_p2].get("ids_enfants",[])
            if id not in l :
                l.append(id)
                etat_civil[id_p2]['ids_enfants'] = l
```

(3.2) *Écrivez une fonction `coparents` qui prend en paramètre un état civil et l'identifiant d'une personne, et qui renvoie la liste (sans doublon) des identifiants des personnes avec qui elle a eu des enfants. (2 pt)*

```python
# Version 1 avec effet de bord : mise à jour des enfants.
def coparents(etat_civil, id_init):
    maj_enfants(etat_civil)
    res = []
    for enfant in etat_civil[id_init].get("ids_enfants", []) :
        id1 = etat_civil[enfant]['id_parent_1']
        id2 = etat_civil[enfant]['id_parent_2']
        if id1 == id_init and id2 not in res :
            res.append(id2)
        if id2 == id_init and id1 not in res :
            res.append(id1)
    return res

#Version 2 sans effet de bord
def coparents(etat_civil, id_init):
    res = {}
    for personne in etat_civil.values() :
        id1 = personne['id_parent_1']
        id2 = personne['id_parent_2']

        if id1 == id_init and id2 != None :
            res[id2] = None
        if id2 == id_init and id1 != None :
            res[id1] = None

    return list(res.keys())
```

*(3.3) Écrivez une fonction `ancetres` qui prend en paramètre un état civil et un id, et renvoie la liste sans doublon des ids de tous ses ancêtres (parents, grands parents, grand-grands parents, ...). (3pts)*

```python
# Version récursive
def ancetres(etat_civil, id):
    res = []
    id1 = etat_civil[id].get('id_parent_1', None)
    id2 = etat_civil[id].get('id_parent_2', None)
    if id1 != None:
        res = res + ancetres(etat_civil, id1) + [id1]
    if id2 != None:
        res = res + ancetres(etat_civil, id2) + [id2]
    return res


# Version itérative
def ancetres(etat_civil, id):
    parents = []
    Q = [id]
    while len(Q)>0 :
        id = Q.pop(0)

        id1 = etat_civil[id].get('id_parent_1', None)
        if id1 != None and id1 not in parents :
            parents.append(id1)
            Q.append(id1)

        id2 = etat_civil[id].get('id_parent_2', None)
        if id2 != None and id2 not in parents :
            parents.append(id2)
            Q.append(id2)

    return parents
```