Polynésie – 2025 – sujet2 - Correction

Exercice 3 (8 points)

Partie A – Structures et graphe

1. Décrire le fonctionnement d'une Pile

Une **pile** est une structure de type **LIFO** (*Last In, First Out*).

- On ajoute un élément au sommet → **empiler**
- On retire l'élément au sommet → **dépiler**
- Le dernier élément ajouté est le premier retiré.

Exemple : pile d'assiettes.

---

2. Décrire le fonctionnement d'une File

Une **file** est une structure de type **FIFO** (*First In, First Out*).

- On ajoute un élément à la fin → **enfiler**
- On retire l'élément en tête → **défiler**
- Le premier élément ajouté est le premier retiré.

Exemple : file d'attente.

---

3. Pourquoi un graphe non orienté ?

Si un mot A est voisin d'un mot B, alors B est aussi voisin de A.

👉 La relation de voisinage est **symétrique**.
Il n'y a donc pas de notion de sens → graphe **non orienté**.

---

4. Graphe pour : gars, mars, mors, ours, purs

Voisinages donnés :

- mars ↔ gars
- mars ↔ mors
- mors ↔ ours
- ours ↔ purs

Représentation :

```
gars — mars — mors — ours — purs
```

👉*Graphe en chaîne.*

Partie B – Fonctions Python

5. Fonction `chaine_vers_tab`

```
def chaine_vers_tab(mot):
    tab_lettres = []
    for lettre in mot:
        tab_lettres.append(lettre)
    return tab_lettres
```

👉*Commentaire :*

- *On crée une liste vide.*
- *On ajoute chaque caractère de la chaîne.*
- *Exemple :* `'ours'` → `['o','u','r','s']`.

---

6. Pourquoi la fonction distance fonctionne ?

```
def distance(mot1, mot2):
    tab = chaine_vers_tab(mot1)
    for lettre in mot2:
        if lettre in tab:
            tab.remove(lettre)
    return len(tab)
```

👉 **Explication :**

- On transforme `mot1` en liste.
- Pour chaque lettre de `mot2`, si elle existe dans la liste, on la retire.
- Il reste uniquement les lettres différentes.
- La longueur restante correspond au nombre de lettres à modifier.

Donc on calcule bien la **distance minimale** entre les deux mots.

---

7. Fonction `renvoie_voisins`

```
def renvoie_voisins(mot):
    tab_voisins = []
    for voisin_possible in TAB_MOTS:
        if distance(mot, voisin_possible) == 1:
            tab_voisins.append(voisin_possible)
    return tab_voisins
```

👉 *Commentaire :*

- *On parcourt tous les mots.*
- *Si la distance vaut 1 → voisin.*
- *On ajoute à la liste.*

Partie C – Recherche du plus court chemin (BFS)

La fonction `dic_parent` réalise un **parcours en largeur (BFS)** grâce à une file.

8. Suite du déroulement pas à pas

On reprend après le 2e tour :

Troisième tour:

- On défile **mors**
- Voisins : mars, ours
- mars est déjà dans parent
- ours n'y est pas → on l'ajoute

parent devient :

```
{
'mars': None,
'gars': 'mars',
'mors': 'mars',
'ours': 'mors'
}
```

file_voisins contient : `ours`

Quatrième tour:

- On défile **ours**
- C'est le mot final → la boucle s'arrête.

La fonction renvoie :

```
{
'mars': None,
'gars': 'mars',
'mors': 'mars',
'ours': 'mors'
}
```

---

9. Fonction `renvoie_pile`

```
def renvoie_pile(parent, mot_final):
    ma_pile = Pile()
    mot = mot_final
    while mot != None:
        ma_pile.empiler(mot)
        mot = parent[mot]
    return ma_pile
```

👉*Commentaire :*

- *On remonte grâce au dictionnaire.*
- *On empile depuis le mot final jusqu'au départ.*
- *Le sommet sera le mot de départ.*

---

10. Fonction `construit_chemin`

```
def construit_chemin(ma_pile):
    tab = []
    while not ma_pile.est_vide():
        mot = ma_pile.depiler()
        tab.append(mot)
    return tab
```

👉*Commentaire :*

- *On dépile successivement.*
- *On reconstruit le chemin dans le bon ordre.*

---

11. Fonction `chercher_chemin`

```
def chercher_chemin(mot_depart, mot_final):
    parent = dic_parent(mot_depart, mot_final)
    pile = renvoie_pile(parent, mot_final)
    return construit_chemin(pile)
```

👉*Commentaire :*

1. *On calcule le dictionnaire des parents (BFS).*
2. *On reconstruit le chemin en pile.*
3. *On transforme en tableau final.*
