Mon objectif avec Glouton2

J'ai voulu créer un algorithme glouton qui affecte des tâches à des secouristes en utilisant une matrice d'adjacence. Mon but était de choisir les meilleures paires possible, étape par étape, en me basant sur une stratégie simple : affecter les tâches les moins flexibles aux secouristes les moins sollicités.

Structure de la matrice tabAffect

J'ai construit la matrice tabAffect de taille 2n x 2n, où :

- Les n premières lignes et n premières colonnes représentent les tâches.
- Les n dernières colonnes (de n à 2n-1) représentent les secouristes.
- Une cellule avec 1 en [i][j] indique que la tâche i peut être faite par le secouriste j-n.

Mon code

1. Validation des entrées

Je commence par vérifier si le tableau est nul, vide, ou s'il n'a pas une taille paire. Comme j'ai besoin de n tâches et n secouristes, le tableau doit avoir un nombre pair de lignes/colonnes :

```
if (tabAffect == null || tabAffect.length == 0 || tabAffect.length % 2 != 0) {
    throw new IllegalArgumentException("Le tableau n'a pas la bonne forme");
```

2. Calcul des degrés

Ensuite, je calcule :

- Le **degré sortant** de chaque tâche (combien de secouristes peuvent la faire),
- Le **degré entrant** de chaque secouriste (combien de tâches il peut faire).

```java
// Calcul des degrés
for (int i = 0; i < n; i++) {
    for (int j = n; j < total; j++) {
        if (tabAffect[i][j] == 1) {
            degreSortant[i]++;
            degreEntrant[j - n]++;
        }
    }
}
```

3. Stratégie gloutonne

Pour chaque tâche à affecter :

- Je cherche la **tâche non encore affectée** qui a le **plus petit degré sortant** (la moins flexible).
- Ensuite, parmi les secouristes disponibles compatibles avec cette tâche, je choisis celui avec le **plus petit degré entrant** (le moins sollicité).

```java
boolean[] tacheAffectee = new boolean[n];
boolean[] secouristeAffecte = new boolean[n];

for (int count = 0; count < n; count++) {
    int minDegTache = Integer.MAX_VALUE;
    int minTache = -1;

    for (int i = 0; i < n; i++) {
        if (!tacheAffectee[i] && degreSortant[i] < minDegTache && degreSortant[i] > 0) {
            minDegTache = degreSortant[i];
            minTache = i;
        }
    }

    int minDegSecouriste = Integer.MAX_VALUE;
    int minSecouriste = -1;

    if (minTache != -1) {
        for (int j = 0; j < n; j++) {
            if (!secouristeAffecte[j] && tabAffect[minTache][j + n] == 1 && degreEntrant[j] < minDegSecouriste) {
                minDegSecouriste = degreEntrant[j];
                minSecouriste = j;
            }
        }
    }

    if (minTache != -1 && minSecouriste != -1) {
        this.affectation.put("Tache " + (minTache + 1), "Secouriste " + (minSecouriste + 1));
        tacheAffectee[minTache] = true;
        secouristeAffecte[minSecouriste] = true;
    }
}
```

Si je trouve une paire valide, je l'ajoute à la HashMap affectation, et je marque la tâche et le secouriste comme affectés. Pour finit par return affectation