Tank Allocation

Introduction aux données

Voici une partie des données de `./data/dataTankAllocation/0000.json`

```
"volumes": [1861, 2262, 256, 0, 0, 1046, 0, 717, 603, 807, 717, 555, 1672, 1606, 2979],
"conflicts": [
        [0, 3],[0, 4], [1, 3], [1, 7], [2, 6], [2, 8],[3, 5], [3, 6], [3, 13], [4, 6], [4, 7], [4, 11], [6, 8], [6, 14], [7, 8], [7, 11], [8, 13], [9, 11]],
"tanks": [
    {
        "capacity": 813,
        "impossibleCargos": [13],
        "neighbors": [11, 16, 6, 9, 23, 17]
    },{
        "capacity": 814,
        "impossibleCargos": [],
        "neighbors": [6, 20, 14, 13, 18]
    }
]
```

- volumes : le volume des marchandises
- conflicts : les conflits entre les marchandises
- tanks
    - capacity : la capacité du tank
    - impossibleCargos : les numéros de séquence des volumes qui ne peuvent pas être chargés
    - neighbors : tanks adjacents, pour éviter que des tanks adjacents contiennent des marchandises en conflit

Contraintes du problème

- C1: Toutes les marchandises dans `volumes` doivent être chargées dans les `tanks`, et $\sum \text{tank} \geq \text{volume}[i]$
- C2: Les marchandises `ImpossibleCargos` ne peuvent pas être chargées dans ce `tank`
- C3: Les marchandises ayant des `Conflicts` ne peuvent pas être chargées dans des `tanks` adjacents

Analyse du code Python

Paramètres de la classe

```
self.path # chemin du fichier de données
self._content # toutes les données
self._volumes # informations sur les volumes
self._conflicts # informations sur les conflits
self._tanks # informations sur les tanks
```

Fonctions de la classe

```
get_new_file(new_path) # efface les données originales et sélectionne un nouveau fichier de données
read_file() # lit les données et les insère dans les paramètres de la classe
define_vars() # définit les variables
def_sat() # définit les contraintes
vefierication(thesolution) # vérifie si la solution satisfait aux contraintes de capacité
solve() # résout le problème
```

Fonction `define_vars`

Définition des paramètres

```
def define_vars(self):
    global vars
    vars = VarArray(
        size=len(self._tanks), # nombre de paramètres = nombre de tanks
        dom=range(len(self._volumes) + 1) # domaine des paramètres, comme il existe volume=0, on peut ajouter + 1, indiquant que cette marchandise n'a pas besoin d'être chargée par un tank
    )
```

Fonction `def_sat`

C1: La capacité totale des tanks qui chargent la marchandise i est supérieure à volumes[i]

```
[
    Sum([self._tanks[i]['capacity'] * (vars[i] == t) for i in range(len(self._tanks))]) >= self._volumes[t]
    for t in range(len(self._volumes)) if self._volumes[t] > 0
],
```

C2: Le tank i ne peut pas charger les marchandises dans impossibleCargos

```
[
    vars[i] != cargo
    for i in range(len(self._tanks))
    for cargo in self._tanks[i]['impossibleCargos']
],
```

C3: Les marchandises en conflit ne peuvent pas être placées dans des tanks adjacents

```
[
    (vars[i] != x) | (vars[j] != y)
    for (x,y) in self._conflicts
    for i in range(len(self._tanks))
    for j in self._tanks[i]['neighbors']
]
```

Fonction `verification`

```
def vefierication(self, thesolution):
    # test de capacité
    tank_volumes = [0 for _ in range(len(self._volumes))] # initialisation de la liste des volumes chargés dans les tanks
    for idx, num in enumerate(thesolution):
        tank_volumes[num] += self._tanks[idx]['capacity'] # somme des capacités de chaque tank qui charge ce numéro
    print(tank_volumes) # affiche la capacité des marchandises
    print(self._volumes) # affiche la capacité des tanks qui chargent les marchandises
    for i,j in zip(tank_volumes,self._volumes):
            if i < j:
                print(f"volumes {tank_volumes.index(i)} is not enough")
                return
        print("Volumes Constraint is satisfied")
        # test impossibleCargos
        for idx, num in enumerate(thesolution):
            for cargo in self._tanks[idx]['impossibleCargos']:
                if num == cargo:
                    print(f"tank {idx} has cargo {cargo} which is impossible")
                    return
        print("impossibleCargos Constraint is satisfied")
        # test de conflit
        for idx, num in enumerate(thesolution):
            for neighbor in self._tanks[idx]['neighbors']:
                neighbor_num = thesolution[neighbor]
```

```
                # vérifier si (num, neighbor_num) ou (neighbor_num, num) existe dans la liste des conflits
                if (num, neighbor_num) in self._conflicts or (neighbor_num, num) in self._conflicts:
                    print(f"Conflict found: tank {idx}(cargo {num}) and tank {neighbor}(cargo {neighbor_num})")
                    return
                    break
        print("Conflict Constraint is satisfied")
```

Comparaison des stratégies

option	time	solved
FrbaOnDom	11.97s	Yes
FirstFail	4.77s	No
MaxDegree	4.41s	No
MinDomain	3.72s	No
DomOverDeg	4.18s	No
Random	5.64s	No
Min	3.86s	No
Max	3.88s	No
Random	5.61s	No
OccurMost	3.89s	No
OccurLeast	3.78s	No

Problèmes rencontrés

1. Impossible de trouver une solution sans ajouter `options="-varh=FrbaOnDom"`
2. Après avoir ajouté d'autres options, il est impossible de trouver une solution, mais la vitesse est plus rapide que `FrbaOnDom`
3. Pour les marchandises avec `volume=0`, une contrainte devrait pouvoir être ajoutée, mais aucune méthode efficace n'a été trouvée, donc la partie `+ 1` dans `dom=range(len(self._volumes) + 1)` n'est pas utilisée efficacement