Journal de Bord

Alexis GIBERT UPSSITECH SRI 2A

Objectifs

R2D2 est un robot placé dans un monde 2D représenté par un graphe non orienté : les arêtes représentent les routes que R2D2 peut suivre, alors que les sommets représentent les lieux où R2D2 a des choses à faire. Les arêtes seront pondérées pour représenter la longueur du chemin que doit parcourir R2D2 pour aller du sommet origine de l'arête au sommet arrivée de l'arête. Et chaque sommet est pondéré par sa position dans le plan 2D du monde (coordonnées euclidiennes).

On prendra comme hypothèse que R2D2 connaît le monde dans lequel il est placé. Le travail que doit faire R2D2 : déposer 1 cube de couleur à chaque lieu de manière à ce qu'il y ait dans deux lieux voisins (c-à-d liés par une arête) des cubes de couleur différente. On considérera que R2D2 dispose de suffisamment de cubes.

Le projet est découpé en 5 étapes réparties sur 12 séances de TP.

1. **Étape 1 (TP 1 et 2):** Résolution de la tâche 1 - Utilisation de la logique propositionnelle pour déterminer si trois couleurs sont suffisantes. Utilisation du solver SAT.
2. **Étape 2 (TP 3 à 6):** Résolution de la tâche 2 - Calcul des chemins les plus courts entre lieux. Trois cas sont abordés, chacun utilisant des algorithmes différents.
    - **Cas 1 :** Calcul du plus court chemin entre deux lieux en utilisant les distances et les coordonnées cartésiennes.
    - **Cas 2 :** Recherche du chemin le plus court passant une fois par chaque lieu et revenant au point de départ.
    - **Cas 3 :** Intégration de la capacité de vol de R2D2, permettant des déplacements en ligne droite entre les lieux.
3. **Étape 3 (TP 7 et 8):** Suite de la tâche 2 - Optimisation des méthodes pour calculer des chemins moins optimaux mais plus rapidement.
4. **Étape 4 (TP 9 et 10):** Résolution de la tâche 3 - Utilisation d'un graphe de contraintes pour déterminer le nombre minimal de couleurs nécessaires.
5. **Étape 5 (TP 11 et 12):** Suite et fin de la tâche 2 - Résolution du cas 3 de la tâche 2 en exprimant le problème à l'aide de formules mathématiques (utilisation de ZIMPL et SCIP).

Début séance TP1 - KUPR7AB1 : (TP1-G1) IA (MCL)-U3-Salle 201 Tuesday, September 19, 2023 / 15:45 - 17:45

Etape 1 : Coloration de Graphes

Rappel du problème

Vous devrez identifier le nom et le type du problème, proposer un encodage en logique propositionnelle et le résoudre en utilisant le solver SAT fourni. Pour cela vous compléterez la classe Etape1 du package etape1 et vous l'exécuterez. Il peut être souhaitable de compléter les tests déjà proposés.

Type du Problème

Problème de Coloration de Graphes, une tâche classique en informatique théorique consistant à attribuer des couleurs à des sommets d'un graphe de manière à ce que des sommets adjacents n'aient pas la même couleur.

Encodage en Logique Propositionnelle

1. **Variables Propositionnelles :**

   Les variables propositionnelles représentent les différentes options de couleur pour chaque lieu du graphe. Chaque variable est un entier unique calculé en fonction du numéro du noeud et du numéro de couleur.

   Exemple

Valeur codée	numéro du noeud	numéro de la couleur
4003	4	003

   On notera que ce codage limite le nombre de couleur à `999` (largement suffisant dans le cadre de nos tests) mais, en contrepartie, permet de ne pas limiter le nombre de noeud.

2. **Clauses pour les Couleurs Possibles pour Chaque Noeud :**

   Les couleurs possibles pour chaque noeud sont représentées par des clauses. Elles expriment la possibilité des différentes couleurs pouvant être prise par un noeud donné.

   - **Exemple :** Si le noeud 1 a 4 couleurs possibles, la clause serait (pour chaque couleur) : (1001, 1002, 1003, 1004).

3. **Clauses pour les Liens entre les Noeuds (Contraintes d'Arêtes) :**

   Les différentes arrêtes du graphe sont représentées par des clauses qui expriment les liens entre deux noeuds donnés.

   - **Exemple :** Si le noeud 1 et le noeud 2 sont connectés par une arête, tout comme le noeud 1 et le noeud 3, la clause serait ("x" : la couleur n'importe pas) : (-100x, -200x), (-100x, -300x)

Début séance TP2 - KUPR7AB1 : (TP2-G1) IA - (MCL)-U3-Salle 201 Thursday, September 21, 2023 / 13:30 - 15:30

4. **Clauses pour les Contraintes de Couleur sur les Arêtes :**

   Les contraintes de couleur sur les arêtes sont représentées par des clauses qui expriment l'impossibilité d'avoir deux fois la même couleur entre les noeuds connectés par une arête.

   - **Exemple :** Si le noeud 1 et le noeud 2 sont connectés par une arête, tout comme le noeud 1 et le noeud 3, et vous avez 4 couleurs, la clause serait (pour chaque couleur) : (-1001, -2001), (-1002, -2002), (-1003, -2003), (-1004, -2004).

5. **Mise en oeuvre dans le Code :**

   Dans le code, ces clauses sont générées dynamiquement en utilisant des boucles et des structures de données pour représenter les différents éléments du graphe. La méthode `updateBase` est responsable de la génération des clauses en fonction du nombre de couleurs spécifié.

   - Les variables sont calculées et stockées dans des listes triés (`color`, `node`, `edge`) en fonction du nombre de couleurs et du nombre de noeuds.
   - Ces listes triés (pour améliorer le débogage du code) sont ensuite incluses dans la base des clauses (`base`), qui est utilisée pour résoudre le problème avec le solver SAT.

Début séance TP3/TP4 - KUPR7AB1 : (TP3 et 4-G1) IA (MCL) -U2-Salle 209 Wednesday, October 11, 2023 - 13:30 - 17:30

6. **Utilisation du SolverSAT :**

   Utilisation de la méthode `solve`. `SolverSAT.solve(self.base)` > Si le solveur renvoie True, la base de clauses est satisfaisable, sinon elle ne l'est pas. En d'autres termes, ça nous indique si le robot peut accomplir sa tâche avec le nombre de couleurs choisi.

7. **Vérification des résultats attendus :**

```
10
town10 with 3 colors (expecting True):  True
Base de clause utilise  30  variables et contient les clauses suivantes :
town10 with 2 colors (expecting False):  False
town10 with 4 colors (expecting True):  True
20
flat20_3_0.col with 4 colors (expecting True):  True
flat20_3_0.col with 3 colors (expecting True):  True
flat20_3_0.col with 2 colors (expecting False):  False
80
jean.col with 10 colors (expecting True):  True
```

```
jean.col with 9 colors (expecting False):  False
jean.col with 3 colors (expecting False):  False
```

Note : Pour deboger on peut afficher les clauses avec `displayBase`.

Exemple :

```
[1001, 1002, 1003]
[2001, 2002, 2003]
...
[7001, 7002, 7003]
[8001, 8002, 8003]
[9001, 9002, 9003]
[10001, 10002, 10003]
[-1001, -2001]
[-1001, -3001]
...
[-10002, -9002]
[-10003, -8003]
[-10003, -9003]
```

Étape 2 - Cas 1 : Recherche des Chemins les Plus Courts

Rappel du Problème

R2D2 cherche à déterminer les chemins les plus courts entre deux lieux de son monde en utilisant les distances entre les lieux et les coordonnées cartésiennes de chaque lieu. Le problème est modélisé en tant que tâche de recherche de chemin optimale.

Nom et Type du Problème

Problème du Plus Court Chemin (PCC) consistant à trouver le chemin le plus court entre deux noeuds d'un graphe pondéré.

Choix du Solveur

J'ai choisi le SolverAStar car c'est un algorithme de recherche de plus court chemin qui utilise une combinaison de coût réel et d'une estimation heuristique du coût restant pour guider la recherche vers la solution optimale.

Modélisation du Problème

La classe `EtatCas1` est responsable de la modélisation du problème du Plus Court Chemin (PCC) dans le contexte spécifique du Cas 1. Chaque fonction remplit un rôle spécifique dans la représentation et la résolution du problème du Plus Court Chemin.

1. **Fonction `estSolution(self)`**

Vérifie si l'état courant est une solution au problème du PCC. Elle renvoie `True` si l'état courant est égal à l'état final, indiquant ainsi que le chemin optimal a été trouvé. Sinon, elle renvoie `False`.

2. **Fonction `successeurs(self)`**

   Génère une liste d'états successeurs à l'état courant. Pour chaque voisin du lieu représenté par l'état courant, un nouvel état est créé. Ces états successeurs représentent les différentes options de mouvement à partir de l'état actuel.

3. **Fonction `h(self)`**

   Calcule l'heuristique de l'état courant. Dans le contexte du PCC, l'heuristique est généralement la distance estimée entre l'état courant et l'état final. Plus cette distance est petite, plus l'heuristique est optimiste.

4. **Fonction `k(self, e)`**

   Calcule le coût du passage de l'état courant à un état `e` donné. Dans le cas du PCC, cela correspond au coût de l'arête entre les deux lieux représentés par ces états.

5. **Fonction `displayPath(self, pere)`**

   Affiche le chemin qui a mené à l'état courant en utilisant la map des pères (`pere`). Elle remonte de l'état final à l'état initial en suivant les liens établis par la map des pères et affiche le chemin optimal trouvé.

6. **Fonctions de Comparaison et de Hachage**

   Les fonctions `__hash__` et `__eq__` sont implémentées pour permettre l'utilisation de la classe dans des structures de données telles que des tables de hachage. Ces fonctions sont nécessaires pour garantir la cohérence lors de la comparaison et du stockage d'objets de cette classe.

Tableau Comparatif des Résultats

Nb Villes	Chemin	Poids Attendu	Poids Obtenu	Nb d'États Explorés	Nb d'États Générés
10	0 à 9	1190.97	1190.97	7	21
10	5 à 9	858.62	858.62	5	16
10	2 à 9	1090.64	1090.64	10	31
10	1 à 7	889.19	889.19	5	14
26	0 à 25	1856.5	1856.5	20	76
146	0 à 145	1143.0	1143.0	150	1327
998	0 à 997	726.7	726.7	1000	44862

Les résultats obtenus correspondent aux attentes, indiquant que le SolverAStar a réussi à trouver les chemins les plus courts dans chaque cas.

Les nombres d'états explorés et générés varient en fonction de la complexité du problème, mais restent dans des limites raisonnables, ce qui montre la réelle efficacité de l'algorithme dans la recherche de chemins optimaux.

D'ailleurs la vitesse d'execution du programme est assez fulgurante (< 1s)

Étape 2 - Cas 2 : Chemins/Cycles les Plus Courts par la terre

Rappel du problème

Le Cas 2 consiste à trouver le chemin le plus court qui passe par chaque lieu une seule fois puis revient au point de départ. Il s'agit d'un problème classique connu sous le nom de Problème du Voyageur de Commerce (TSP).

Modélisation du problème

1. **Méthode `successeurs(self)`**

   La méthode `successeurs` est cruciale pour ce problème, car elle détermine les mouvements possibles à partir de l'état actuel. Voyons comment elle est implémentée et comment elle contribue à la recherche de la solution :

   1. **Récupération des Voisins Non Visités :** La méthode commence par récupérer le dernier point visité dans le chemin (`dernier_point`). Ensuite, elle itère sur tous les points du graphe et sélectionne ceux qui n'ont pas encore été visités (`point not in self.etats_visites`) et qui ne sont pas égaux au dernier point (`point != dernier_point`).
   2. **Création des États Successeurs :** Pour chaque point sélectionné, un nouvel état est créé. Ce nouvel état est une copie de l'état actuel, mais avec le nouveau point ajouté au chemin. De plus, le nouveau point est ajouté à l'ensemble `etats_visites`.
   3. **Stockage dans une Liste :** Les états successeurs ainsi générés sont stockés dans une liste, qui est ensuite renvoyée par la méthode.

2. **Méthode `k(self, e)`**

   La méthode `k` est responsable du calcul du coût du passage de l'état actuel à l'état `e`. Dans le cas du TSP, le coût est simplement le coût de l'arête entre le dernier point du chemin actuel et le premier point du chemin de l'état `e`. Cela permet de garantir que le chemin revient au point de départ.

3. **Méthode `calculerPoids(self)`**

   Cette méthode calcule le poids total du chemin en additionnant les coûts de toutes les arêtes du chemin. Elle est utilisée pour afficher le poids total

du chemin dans la méthode `displayPath`.

Identification d'un problème

Les résultats obtenus ne correspondent pas aux attentes pour deux raisons principales :

1. **Poids Incorrect** : Le poids calculé pour le chemin obtenu est différent de celui attendu, ce qui indique un problème dans le calcul du coût total.
   - Attendu : `3792.190362007193`
   - Obtenu : `1745.2388470688104`
2. **Chemin Incorrect** : Le chemin obtenu ne correspond pas au chemin attendu, suggérant une erreur dans la génération des successeurs ou dans le suivi du chemin parcouru.
   - Attendu : `[0, 1, 3, 4, 8, 9, 7, 6, 5, 2, 0]`
   - Obtenu : `[0, 3, 0, 1, 2, 4, 5, 6, 7, 8, 9]`

Début séance TP5 KUPR7AB1 : (TP5-G1) IA (MCL)-U2-Salle 218 Thursday, November 16, 2023 / 10:00 - 12:00

Correction du problème

Le problème dans le code résidait dans la manière dont les états visités étaient gérés. Le chemin était mis à jour en ajoutant le point courant à chaque étape, et les états visités étaient stockés dans un ensemble ce qui conduisait à une mauvaise représentation de l'état et à des erreurs dans la génération des successeurs.

Dans le code corrigé, j'ai apporté plusieurs modifications pour résoudre ces problèmes :

1. **Utilisation d'une Liste pour les États Visités :** Au lieu d'utiliser un ensemble pour stocker les états visités, j'ai utilisé une liste. Cela permet une meilleure gestion de l'ordre des états visités et de la vérification de l'état final.
2. **Correction de la Méthode `__eq__` :** Dans la méthode `__eq__`, la comparaison était basée sur les états visités (`self.etats_visites == o`). J'ai corrigé cela en comparant les états courants (`self.etat_courant == o`).
3. **Utilisation du Paramètre `etat_debut` :** J'ai introduit un nouveau paramètre `etat_debut` dans le constructeur pour stocker le point de départ initial. Cela est nécessaire pour vérifier si le retour au point de départ est possible à la fin.
4. **Modification de la Méthode `__hash__` :** La méthode `__hash__` a été modifiée pour utiliser un tuple de la liste d'états visités plutôt que la liste elle-même, ce qui permet d'obtenir un hashable.

Début séance TP6 KUPR7AB1 : (TP6-G1) IA (MCL) - U3-Salle 207 Tuesday, November 21, 2023 / 13:30 - 15:30

5. **Mise à Jour de la Méthode `successeurs` :** La méthode `successeurs` a été mise à jour pour prendre en compte le retour au point de départ une fois tous les lieux visités.

Tableau Comparatif des Résultats

CAS 2	**Résultat Attendu**	**Résultat Obtenu**
Sur 10 villes	[0, 1, 3, 4, 8, 9, 7, 6, 5, 2, 0]	[0, 1, 3, 4, 8, 9, 7, 6, 5, 2, 0]
Longueur du chemin	3792.190362007193	3792.190362007193
Nombre d'états explorés	-	330
Nombre d'états générés	-	341

Globalement, l'algorithme produit des résultats corrects avec une efficacité raisonnable en termes de nombre d'états explorés et générés.

Début séance TP7/8 KUPR7AB1 : (TP7 et 8-G1) IA (MCL)- U3-Salle 214 Wednesday, November 22, 2023 / 08:00 - 12:00

Etape 2 - Cas 3 : Chemins/Cycles les Plus Courts par le vol

Rappel du problème

R2D2 vient d'être "upgradé" : son concepteur l'a équipé de la capacité à voler. Il peut désormais relier en ligne droite chacun des lieux de son monde sans être obligé de suivre les routes. Il peut donc trouver un autre chemin plus court permettant de passer par chaque lieu et de revenir ensuite à son point de départ.

Modélisation/Adaptation du problème à partir d'`EtatCas2`

Fonction	`EtatCas2`	`EtatCas3`
Constructeur	`tg`, `etat_visit=None`, `etat_courant=0`, `etat_debut=0`	`tg`, `etat_visit=None`, `etat_courant=0`, `etat_debut=0`

Fonction	`EtatCas2`	`EtatCas3`
estSolution	Condition basée sur la longueur des sommets visités et l'égalité entre le début et la fin	Condition basée sur la longueur des sommets visités et l'égalité entre le début et la fin
successeurs	Construction de successeurs basés sur les adjacents non visités	Construction de successeurs basés sur les sommets non visités
h	Heuristique basée sur la distance restante à parcourir par voie terrestre (`self.tg.getPoidsMinTerre()`)	Heuristique basée sur la distance restante à parcourir par voie des airs (`self.tg.getPoidsMinAir()`)
k	Coût basé sur `self.tg.getCoutArete`	Coût basé sur `GrapheDeLieux.dist`
displayPath	Affiche les sommets visités	Affiche le chemin trouvé en utilisant les sommets visités
hash	Basé uniquement sur les sommets visités (`tuple(self.etat_visit)`)	Prend en compte également les coordonnées cartésiennes des sommets (`hash((self.tg, self.etat_courant, tuple(self.etat_visit)))`)
eq	Comparaison basée sur les sommets visités	Comparaison basée sur les sommets visités
str	Représentation sous forme de chaîne des sommets visités	Représentation sous forme de chaîne des sommets visités

Tableau Comparatif des Résultats

Nb Villes	Résultat Attendu	Résultat Obtenu	Nb d'États Explores	Nb d'États Générés
6	1360.6495955560758 [0, 1, 2, 3, 5, 4, 0]	1360.6495955560758 [0, 1, 2, 3, 5, 4, 0]	131	238
7	1638.459980067224 [0, 1, 2, 3, 6, 5, 4, 0]	1638.459980067224 [0, 1, 2, 3, 6, 5, 4, 0]	552	1110
8	1729.6228205017967 [0, 4, 5, 7, 6, 3, 2, 1, 0]	1729.6228205017967 [0, 4, 5, 7, 6, 3, 2, 1, 0]	1430	3514
9	1855.2162397331167 [0, 1, 2, 3, 6, 7, 5, 8, 4, 0]	1855.2162397331167 [0, 1, 2, 3, 6, 7, 5, 8, 4, 0]	4796	13342

Nb Villes	Résultat Attendu	Résultat Obtenu	Nb d'États Explores	Nb d'États Générés
10	2026.2675322208256 [0, 1, 2, 3, 6, 7, 5, 8, 9, 4, 0]	2026.2675322208256 [0, 1, 2, 3, 6, 7, 5, 8, 9, 4, 0]	16052	50412

Note :

- Les poids attendus et obtenus concordent pour les cas testés, confirmant que l'algorithme produit le poids correct.
- Les chemins attendus et obtenus concordent pour les cas testés, montrant que l'algorithme génère le chemin optimal.

Comme les résultats sont convaincant la prochaine séance je pourrai directement commencer l'étape 3.

Séance TP9/10 KUPR7AB1 : (TP9 et 10-G1) IA (MCL)-U3-Salle 214 Wednesday, November 29, 2023 / 08:00 - 12:00

Etape 3 - Plus rapide !

Rappel du problème

R2D2 se rend compte que sa méthode précédente met trop de temps à s'exécuter ! Du coup, il renonce à trouver le chemin le plus court et est prêt à tenter des chemins un peu moins bons pourvu qu'il arrive à les calculer plus vite. Et comme il est curieux, il va essayer deux méthodes différentes pour voir celle qui est la plus efficace. Vous devrez proposer un mode de représentation et résoudre le problème en utilisant au moins deux des algorithmes fournis. Pour cela vous compléterez la classe UneSolution du package etape3 et vous l'utiliserez pour compléter et exécuter la classe Etape3 du package etape3. Il peut être souhaitable de compléter les tests déjà proposés.

Modélisation du Problème

Le problème que nous cherchons à résoudre est le problème du voyageur de commerce (TSP), qui consiste à trouver le chemin le plus court passant par toutes les villes d'un graphe pondéré. Dans notre cas, le graphe est représenté par la classe `GrapheDeLieux` et les villes sont les sommets du graphe.

J'ai choisi de représenter une solution par une liste ordonnée des indices des villes. Chaque indice représente l'ordre de visite des villes dans le chemin. Par exemple, si la liste est `[0, 2, 1]`, cela signifie que le chemin commence par la ville 0, puis va à la ville 2, et enfin à la ville 1.

La méthode d'évaluation `eval` calcule la longueur totale du chemin en ajoutant les coûts des arêtes entre chaque paire de villes consécutives, ainsi que le coût de l'arête entre la dernière et la première ville.

La méthode `lesVoisins` génère une liste de voisins en échangeant aléatoirement les positions de deux villes dans le chemin pour explorer différentes configurations du chemin.

La méthode `unVoisin` retourne un voisin en choisissant aléatoirement une ville adjacente à la dernière ville du chemin (spécifique à la méthode Hill Climbing).

La méthode `nelleSolution` génère une solution initiale en mélangeant aléatoirement l'ordre des villes.

Choix du solver

Solveur	Type	Avantages	Inconvénients
SolverAStar	Recherche de chemin	Efficace pour trouver le chemin le plus court	Gourmand en ressources pour des graphes grands
SolverCSP	Contraint Satisfaction	Utile pour des problèmes avec contraintes	Peut ne pas être efficace pour la recherche de chemin
SolverSAT	Boolean Satisfiability	Utilisé pour des problèmes NP-complets	Pas toujours adapté à la recherche de chemin
SolverTabou	Metaheuristique	Efficace pour l'optimisation, évite les optima locaux	Ne garantit pas la meilleure solution
SolverHC	Metaheuristique	Simple et rapide, converge souvent vers une solution locale	Peut rester bloqué dans un minimum local, pas de garantie de meilleure solution globale

Pour résoudre le problème du TSP, j'ai choisi d'utiliser deux métaheuristiques : le SolverTabou et le SolverHC (Hill Climbing).

SolverTabou Le SolverTabou est une métaheuristique de recherche locale. Il maintient une liste de solutions tabou pour éviter de revisiter les mêmes solutions et pour sortir des optima locaux. Il explore le voisinage d'une solution en échangeant deux villes dans le chemin. C'est un choix approprié pour notre problème car il est efficace pour l'optimisation, et il évite de rester bloqué dans des optima locaux.

SolverHC (Hill Climbing) Le SolverHC est une métaheuristique de recherche locale également, mais plus simple. Il explore le voisinage d'une solution en échangeant deux villes dans le chemin et accepte le voisin si sa valeur (longueur du chemin) est meilleure que la solution actuelle. Bien qu'il puisse rester bloqué dans un minimum local, il est rapide et souvent converge vers une solution locale acceptable.

En utilisant ces deux métaheuristiques, nous pouvons comparer leur efficacité et voir comment elles se comportent sur différentes instances du problème TSP avec des nombres variés de villes.

Développement de `Etape3` et `UneSolution`

La classe `Etape3` est développée pour tester les deux solveurs sur différentes instances du problème TSP, avec des graphes de différentes tailles (10, 26, 150, 1000 villes). Les résultats sont affichés pour évaluer la performance des solveurs sur ces instances.

Identification d'un problème

L'exécution du programme actuel semble rencontrer un problème de boucle infinie pour le cas avec 10 villes.

```
10
======== Solver 1 pour 10 villes de 0 a 9 :
```

Résultats attendus

```
Etape 3 :
=============
HC, Tabou : sur 10 villes, pas moins de 2026
Sur les autres ???
```

Séance TP10/11 KUPR7AB1 : (TP11 et 12 et fin-G2) IA (MCL) -U2-Salle 211 Tuesday, December 5, 2023 / 13:30 - 17:30

Correction du Problème de Boucle Infinie

Pour résoudre le problème de boucle infinie rencontré, plusieurs modifications ont été apportées à la classe `UneSolution` :

1. **Nouveau Constructeur :** Un nouveau constructeur a été ajouté à la classe `UneSolution` pour permettre la création d'une solution à partir d'un cycle préexistant. Cela a été réalisé en ajoutant un paramètre `cycle` au constructeur, qui représente l'ordre de visite des villes.

   ```
   def __init__(self, tg: GrapheDeLieux, cycle=None):
       # ...
   ```

2. **Initialisation du Cycle :** L'initialisation du cycle a été modifiée pour utiliser le nouveau constructeur et permettre la création d'une solution avec un cycle spécifique ou aléatoire.

```
if cycle == None:
    self.cycle = [i for i in range(self.tg.getNbSommets())]
    random.shuffle(self.cycle)
    self.cycle.append(self.cycle[0])
else:
    self.cycle = cycle
```

3. **Méthode `lesVoisins` :** La méthode `lesVoisins` a été ajustée pour éviter la génération de voisins redondants. Une liste `list_i` est utilisée pour stocker les indices déjà sélectionnés afin d'éviter les doublons.

```
for i in range(self.tg.getNbSommets() - 1):
    index = random.randint(1, self.tg.getNbSommets() - 1)
    index2 = random.randint(index + 1, self.tg.getNbSommets())
    while index in list_i:
        # ...
```

Résulats Obtenu / Résultats Attendus

```
Etape 3 :
=============
HC, Tabou : sur 10 villes, pas moins de 2026
Sur les autres ???
```

En effet, j'obtient des résultats > 2026. Par exemple, pour les 2 premiers cas, on a :

Cas	Solveur	Valeur Optimale	Nombre d'États Explorés
1	Hill Climbing	2500.68	159
1	Tabou	2026.27	100
2	Hill Climbing	6810.99	168
2	Tabou	2803.58	200

Conclusions

Dans les deux cas de test, le Solver 2 (Tabou) semble être plus efficace que le Solver 1 (Hill Climbing). Il produit des solutions de meilleure qualité (valeur plus faible) avec un nombre d'états explorés moindre.

Début des vacances

Etape 4 : Coloration de graphe

Rappel du problème

Finalement, R2D2 cherche à savoir combien il lui faut de couleurs a minima pour réaliser son travail (tâche 3). Vous devrez identifier le nom et le type du problème, proposer un encodage sous la forme d'un graphe de contraintes et le résoudre en utilisant un des algorithmes fournis. Pour cela vous complèterez et exécuterez la classe Etape4 du package etape4. Il peut être souhaitable de compléter les tests déjà proposés

Modelisation du problème

1. **Problème :** Coloration de graphe (Graph Coloring Problem)
2. **Encodage sous forme de graphe de contraintes :** Chaque noeud du graphe représente une zone à colorier, et les arêtes du graphe représentent les liens entre les zones. La contrainte est que deux zones adjacentes ne peuvent pas avoir la même couleur.
3. **Algorithme utilisé :** SolverCSP (algorithme de résolution de problèmes de contraintes)

Tableau Comparatif des Résultats

Test	Attendu	Obtenu
town10.txt avec 3 couleurs	OK avec 3 couleurs	OK avec 3 couleurs, 66 solutions trouvées.
town10.txt avec 2 couleurs	NOK (Pas de solution)	NOK (Pas de solution)
town10.txt avec 4 couleurs	OK avec 4 couleurs	OK avec 4 couleurs, plusieurs solutions trouvées.
flat20_3_0.col avec 4 couleurs	OK avec 4 couleurs	OK avec 4 couleurs, chaque zone est coloriée conformément aux contraintes.
flat20_3_0.col avec 3 couleurs	OK avec 3 couleurs	OK avec 3 couleurs, chaque zone est coloriée conformément aux contraintes.
flat20_3_0.col avec 2 couleurs	NOK (Pas de solution)	NOK (Pas de solution)
jean.col avec 10 couleurs	OK avec 10 couleurs	OK avec 10 couleurs, plusieurs solutions trouvées.
jean.col avec 3 couleurs	NOK (Pas de solution)	NOK (Pas de solution)

Test	Attendu	Obtenu
jean.col avec 9 couleurs	NOK (Pas de solution)	NOK (Pas de solution)

Étape 5 : Résolution du Problème TSP avec SCIP

Rappel du problème

Réalisation de la tâche 2 (suite et fin). Comme R2D2 aime aussi beaucoup les maths et qu'il veut épater ses concepteurs, il reprend le cas 3 de la tâche 2, et cherche à le résoudre en l'exprimant à l'aide de formules mathématiques. Vous devrez proposer un encodage approprié utilisant le langage ZIMPL et résoudre le problème en utilisant le solver SCIP

L'objectif est de trouver le meilleur chemin qui passe par toutes les villes une seule fois et retourne à la ville de départ, minimisant ainsi la distance totale parcourue.

Programme ZIMPL pour n villes

```
import os
os.system("sudo apt-get install libtbb-dev")
os.system("sudo apt-get install libopenblas-base")

import re

# Fonction pour extraire la valeur de "objective value:" d'un fichier de log
def extract_objective_value(file_path):
    with open(file_path, 'r') as file:
        for line in file:
            if "objective value:" in line:
                match = re.search(r'[-+]?\d*\.\d+|\d+', line)
                if match:
                    return float(match.group())
    return None

# Chemin vers libtbb.so.2
tbb_path = '/snap/blender/4300/lib'

# Ajouter le chemin au LD_LIBRARY_PATH
os.environ['LD_LIBRARY_PATH'] = f"{tbb_path}:{os.environ.get('LD_LIBRARY_PATH', '')}"

contenuFichierZIMPL = """
  set V     :=  {1..num_villes};
  set E     := { <i , j > in V * V with i < j };
  set P[]   := powerset (V ) ;
```

```
  set K     := indexset (P ) ;

  param px[V]:= read "../../Data/pb-etape5/tspnum_villes.txt" as "1n" skip 4;
  param py[V]:= read "../../Data/pb-etape5/tspnum_villes.txt" as "2n" skip 4;

  defnumb dist(a, b) := sqrt((px[a] - px[b])^2 + (py[a] - py[b])^2);

  var x [E] binary ;
  minimize cost : sum <i,j> in E : dist (i,j) * x [i,j] ;
  subto two_connected : forall <v> in V do
      (sum <v, j> in E : x [v, j] ) + (sum<i, v> in E : x [i, v] ) == 2;
  subto no_subtour :
    forall <k> in K with card(P[k]) > 2 and card(P[k]) < card(V) - 2 do
        sum <i, j> in E with <i> in P[k] and <j> in P[k] : x[i, j] <= card(P[k]) - 1;
"""

for num_villes in range(6, 20):
    fichier_zpl = f'projet/etape5/town{num_villes}.zpl'
    with open(fichier_zpl, 'w') as f:
        f.write(contenuFichierZIMPL.replace("num_villes", str(num_villes)))
    os.system(f'projet/solvers/SCIP/bin/scip -f {fichier_zpl} > projet/etape5/log{num_villes
    print("Pour ",num_villes," villes  \t",extract_objective_value(f'projet/etape5/log{num_v
```

Résultats Attendus / Obtenus

Nombre de Villes	Poids Approximatif Attendu	Poids Approximatif Obtenu
6	1360	1360.64959555608
7	1638	1638.45998006722
8	1729	1729.6228205018
9	1855	1855.21623973312
10	2026	2026.26753222083
11	2204	2204.34930872984
12	2231	2231.43211218805
13	2247	2247.70309854912
14	2311	2311.70111550665
15	2317	2317.57076714782
16	2353	2353.80139215921
17	2369	2369.65704299013
18	2376	2376.85028163456
19	2405	2405.14241771809

Conclusion

R2D2 a réussi à résoudre le Problème du Voyageur de Commerce pour différentes configurations de villes en utilisant SCIP et ZIMPL. Les résultats obtenus semblent conformes aux attentes, montrant les meilleurs parcours et les poids associés pour chaque cas. Le programme ZIMPL semble bien adapté à la modélisation et à la résolution de problèmes d'optimisation combinatoire comme le TSP. R2D2 peut maintenant épater ses concepteurs avec ces résultats mathématiques impressionnants!