Journal de Bord

Solal Danton Laloy

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 travail de R2D2 va se décomposer en plusieurs tâches :

tâche 1 : Au début, R2D2 décide de n'utiliser que 3 couleurs. Il va donc devoir "raisonner" pour savoir si ces 3 couleurs lui suffisent ou pas pour réaliser son travail.

tâche 2 : Ensuite, R2D2 cherche à savoir comment il va aller déposer les cubes le plus rapidement possible. Pour cela, il cherche des chemins les plus courts en terme de distance parcourue.

tâche 3 : Finalement, R2D2 cherche à savoir combien il lui faut de couleurs a minima pour réaliser son travail

- **Séance TP1**

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.

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.

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.

- **Séance TP2**

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.

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.

Séance TP3/TP4

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 :

```
Resultat obtenu (on attend True) : True
Test sur fichier flat20_3_0.col avec 4 couleurs
20
Resultat obtenu (on attend True) : True
Test sur fichier flat20_3_0.col avec 3 couleurs
Resultat obtenu (on attend True) : True
Test sur fichier flat20_3_0.col avec 2 couleurs
Resultat obtenu (on attend False) : False
Test sur fichier jean.col avec 10 couleurs
80
Resultat obtenu (on attend True) : True
Test sur fichier jean.col avec 9 couleurs
Resultat obtenu (on attend False) : False
Test sur fichier jean.col avec 3 couleurs
Resultat obtenu (on attend False) : False
```

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 qui consiste 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. 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

Nombre	Chemin	Valeur	Valeur max.	Moyenne	Somme
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.

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. Ensuite, elle itère sur tous les points du graphe et sélectionne ceux qui n'ont pas encore été visités et qui ne sont pas égaux au 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. Penser à rajouter l'état de retour.

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`.

- **Séance TP5/6**

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. Méthode `__eq__` :

Dans la méthode `__eq__`, la comparaison est basée sur les états courants (`self.etat_courant == o`).

3. 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.

Tableau Comparatif des Résultats

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

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

- **Séance TP7/8**

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`
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

Séance TP9/10

Etape 3

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. Pour résoudre le problème du TSP, j'ai choisi d'utiliser le SolverTabou et le Solver Hill Climbing.

En utilisant ces deux heuristiques, 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.

Séance TP10

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.

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 compléterez 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)

Séance TP11/12

Étape 5 : Résolution du Problème TSP avec SCIP

Rappel du problème

Réalisation de la tâche 2

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.

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.