UPSSITECH
ÉCOLE D'INGÉNIEURS

SRI 2A
Promotion : 2024-2025

RAPPORT DE TP D'IA

Crampette Olivier

Lien du github :
https://github.com/OlivierCrt

Sommaire :

I - Etape 1

Coloration de graphes:

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

1. Variables Propositionnelles :

Les variables propositionnelles représentent les différentes combinaisons possibles de couleurs pour chaque nœud du graphe. Chaque variable est représentée par un entier unique calculé en fonction du numéro du nœud et du numéro de couleur.

Exemple :

- Valeur codée : 4003
    - Nœud : 4
    - Couleur : 3

2. Clauses pour les Couleurs Possibles pour Chaque Nœud :

Les couleurs possibles pour chaque nœud sont exprimées par des clauses. Chaque clause représente les différentes couleurs qu'un nœud donné peut prendre.

Exemple :
Si un nœud 1 peut avoir 4 couleurs possibles, les clauses seraient :

- (1001, 1002, 1003, 1004)

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

Les liens entre les nœuds (arêtes) sont représentés par des clauses. Ces clauses expriment les relations entre deux nœuds connectés.

Exemple :
Si le nœud 1 est relié au nœud 2 et au nœud 3, alors les clauses seraient :

- (-50c, -600c), (-50c, -400c)

Ici, "c" représente toute couleur et importe peu dans ce contexte.

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

Les contraintes de couleurs sur les arêtes sont exprimées par des clauses. Ces clauses indiquent qu'il est impossible d'avoir la même couleur entre deux nœuds connectés par une arête.

Exemple :
Si le nœud 1 est relié au nœud 2 et au nœud 3, et que 4 couleurs sont disponibles, alors les clauses seraient :

- (-1001, -2001), (-1002, -2002), (-1003, -2003), (-1004, -2004)

5. Code :

Dans le code, ces clauses sont générées à l'aide de boucles et de structures de données représentant les différents éléments du graphe. La méthode majBase est responsable de la création des clauses en fonction du nombre de couleurs spécifié.

- Les variables sont calculées et stockées dans des listes triées (color, node, edge) selon le nombre de couleurs et de nœuds.
- Ces listes triées (pour une meilleure lisibilité et débogage du code) sont ensuite ajoutées à la base de clauses (base). Cette base est utilisée pour résoudre le problème avec un solveur SAT.

On peut utiliser SolverSat.solve() qui renvoie True ou False si c'est satisfiable ou pas.

Exemple de log:

Test sur fichier town10.txt avec 4 couleurs

Base de clause utilise 2 variables et contient les clauses suivantes :

[-1, 2]

[-1, -2]

Resultat obtenu (on attend True) : True

II - Etape 2

TP 3 `a 6 Réalisation de la tache 2. Trois cas sont prévus :
— Cas 1 : R2D2 commence par calculer le plus court chemin entre deux lieux de son monde. Pour cela, il utilise les distances entre les lieux, ainsi que les coordonnées cartésiennes de chaque lieu.
— Cas 2 : Puis il cherche le chemin le plus court permettant de passer dans chaque lieu une seule fois puis de revenir ensuite à son point de départ. 1
— Cas 3 : R2D2 vient d"être "upgradé" : son concepteur l'équipe 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.

Objectif du Cas 1

Dans le Cas 1 de l'Étape 2, R2D2 a pour mission de trouver le chemin le plus court entre deux lieux de son monde. Pour cela, il utilise les distances entre les lieux ainsi que les coordonnées cartésiennes associées à chaque lieu.

Description du Cas 1

Dans ce cas, R2D2 doit :

1. **Initialisation** :
    - Départ d'une ville spécifiée (param1).
    - Destination finale spécifiée (param2).
2. **Calcul du chemin** :
    - R2D2 doit explorer les villes voisines non visitées pour construire le chemin le plus court vers la destination finale.
    - L'objectif est d'utiliser une approche basée sur la distance minimale entre les villes, tout en prenant en compte les coordonnées cartésiennes associées pour une estimation précise des distances.
    -

Description du Module

Classe EtatCas1

La classe EtatCas1 hérite de la classe Etat et est spécifique à la résolution du problème du Voyageur de Commerce (TSP) sous le Cas 1. Voici un résumé des éléments clés de cette classe :

- **Attributs** :
    - tg : Instance de GrapheDeLieux représentant le réseau de lieux et leurs distances.
    - courant : Ville actuellement visitée.
    - final : Destination finale ou dernière ville.
- **Méthodes** :
    - estSolution(self) : Vérifie si l'état courant correspond à la destination finale.
    - successeurs(self) : Retourne les états successeurs basés sur les voisins non visités.
    - h(self) : Calcul de l'heuristique basée sur la distance restante à parcourir.
    - k(self, e) : Calcul du coût du passage à travers un autre état e.
    - displayPath(self, pere) : Affiche le chemin parcouru avec la longueur totale du chemin.
    - __hash__(self) : Code de hachage pour des usages dans des structures de données hashées.
    - __eq__(self, o) : Comparaison d'égalité entre deux états.

- __str__(self) : Représentation sous forme de chaîne de caractères pour l'affichage.

Particularités cas 1:

- **Distances** : Utilisation des distances minimales pour déterminer le chemin le plus court.
- **Coordonnées Cartésiennes** : Intégration des coordonnées pour une précision accrue des distances.
- **Exploration des Successeurs** : Construction des successeurs basés sur des voisins non visités.

Objectif du Cas 2

Dans le Cas 2 de l'Étape 2, R2D2 cherche à trouver le chemin le plus court permettant de visiter chaque lieu une seule fois, puis de revenir à son point de départ. Ce problème correspond au "Voyageur de Commerce avec circuit fermé".

Description du Cas 2

Dans ce cas, R2D2 doit :

1. **Initialisation** :
   - Lieu de départ fixé à la ville 0.
   - Liste des lieux à visiter (a_visiter) fournie, qui contient tous les autres lieux du graphe.
2. **Calcul du chemin** :
   - Trouver un chemin qui visite chaque lieu une seule fois dans l'ordre spécifié, puis revenir au point de départ (ville 0).
   - Le coût du chemin est déterminé par la somme des distances parcourues entre chaque lieu ainsi que le retour au départ.

Description du Module

Classe EtatCas2

La classe EtatCas2 hérite de la classe Etat et est spécifiquement conçue pour résoudre le problème du "Voyageur de Commerce avec circuit fermé" sous le Cas 2. Voici les éléments clés de cette classe :

- **Attributs** :
  - tg : Instance de GrapheDeLieux représentant le réseau de lieux et leurs distances.
  - courant : Lieu actuellement visité.
  - a_visiter : Liste des lieux restants à visiter.
- **Méthodes** :

- estSolution(self) : Vérifie si tous les lieux ont été visités et que l'on est revenu au point de départ.
- successeurs(self) : Retourne les successeurs basés sur les lieux restants à visiter.
- h(self) : Heuristique basée sur le nombre de lieux restants multiplié par le poids minimum d'une arête.
- k(self, e) : Calcul du coût du passage de l'état courant à un autre état.
- displayPath(self, pere) : Affichage du chemin parcouru.
- __hash__(self) : Code de hachage pour utilisation dans des structures de hachage.
- __eq__(self, o) : Comparaison d'égalité entre deux états.
- __str__(self) : Représentation sous forme de chaîne de caractères pour l'affichage.

Particularités:

1. **Circuit Fermé** :
   - Chaque lieu du graphe doit être visité exactement une seule fois.
   - Une fois tous les lieux visités, le chemin doit se terminer en retournant au point de départ (ville 0).
2. Cela signifie que le chemin formé est un circuit fermé, et chaque lieu (excepté le point de départ) doit être traversé uniquement une fois.
2. **Calcul du Coût** :
   - Le coût du chemin est la somme des distances entre chaque paire de lieux visités dans l'ordre.
   - L'heuristique utilisée repose sur le nombre de lieux restants à visiter multiplié par la distance minimale entre deux lieux. Cette approche aide à guider la recherche vers les solutions optimales en minimisant le nombre de lieux à visiter.
3. **Successeurs** :
   - Les successeurs sont les états potentiels suivants à partir de l'état courant.
   - Chaque successeur représente un état où R2D2 se déplace vers un autre lieu qui reste à visiter.
   - Si tous les lieux ont été visités, et que le lieu suivant est le point de départ (ville 0), alors le successeur représente le retour à la base.

Conclusion

Le Cas 2 permet de résoudrele problème du voyageur de commerce avec circuit fermé, où chaque lieu doit être visité une fois, suivi d'un retour au point de départ. On gère gère donc les états correspondants en prenant en compte les distances et les lieux à visiter, tout en optimisant le coût global du chemin.

Description du Cas 3

Dans ce cas, R2D2, le robot explorateur, a été mis à jour avec une capacité spéciale : la capacité de voler. Cette capacité permet à R2D2 de relier directement chaque lieu de son monde, sans se limiter aux routes existantes. Cette nouvelle fonctionnalité lui permet de rechercher le chemin le plus court, passant par chaque lieu, puis de revenir au point de départ.

Classe EtatCas3

Pour gérer ce comportement, une classe EtatCas3 a été développée. Cette classe hérite de la classe Etat et est spécialement conçue pour traiter les transitions entre les différents états tout en tenant compte de la capacité de vol de R2D2.

Explications des Méthodes

1. **init(self, tg, courant=0, a_visiter=None)** :
    - Initialise l'état avec le graphe du monde, la position actuelle de R2D2 et la liste des lieux à visiter.
    - Si le lieu courant fait partie des lieux restants à visiter, il est retiré de la liste.
2. **estSolution(self)** :
    - Vérifie si tous les lieux ont été visités et si R2D2 est retourné au point de départ (lieu 0).
3. **successeurs(self)** :
    - Construit la liste des successeurs en utilisant des distances minimales en ligne droite pour tous les lieux restants. Si tous les lieux sont visités, retourne le successeur du retour au point de départ.
4. **h(self)** :
    - Retourne une heuristique basée sur la distance minimale entre le lieu actuel et les autres lieux restants, optimisant la recherche de solutions courtes.
5. **k(self, e)** :
    - Calcul du coût du passage d'un état à un autre en utilisant les distances directes.
6. **displayPath(self, pere)** :
    - Affiche le chemin suivi en remontant les états depuis le point de départ en utilisant une carte des pères pour chaque transition.

Analyse du Comportement

Graphe :

- Le graphe (GrapheDeLieux) représente les lieux et les distances entre eux, permettant à R2D2 de naviguer efficacement en volant.

Etat :

- Chaque instance de EtatCas3 correspond à une configuration spécifique où R2D2 est situé dans un lieu donné, avec une liste des lieux restants à visiter.

Optimisation :

- Grâce à l'utilisation de distances minimales et à l'heuristique, R2D2 peut rapidement trouver le chemin le plus court en passant par chaque lieu, tout en optimisant ses déplacements grâce à la capacité de vol.

On utilise A star pour chaque cas.

III - Etape 3

TP 7 et 8 Réalisation de la tâche 2 (suite du cas 3). R2D2 se rend compte que sa méthode précédente met trop de temps à s'exécuter ! Du coup, il renonce `a 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.

Objectif

Dans cette partie, R2D2 cherche à optimiser ses méthodes de recherche pour des chemins "raisonnablement bons" tout en les calculant plus rapidement. Deux méthodes différentes sont comparées afin de déterminer laquelle est la plus efficace.

Classe UneSolution

La classe UneSolution hérite de la classe Solution et est spécialement conçue pour traiter le Cas 3 du problème du voyageur de commerce avec la capacité de vol.

Attributs :

- **tg** : Instance de GrapheDeLieux, représentant le monde de R2D2.
- **ordre_visite** : Liste des lieux visités dans l'ordre spécifique.
- **dep** : Départ, point initial de R2D2.

Méthodes principales :

1. **init(self, tg: GrapheDeLieux, depart: int = 0, l_ordre_visite=None)** :
    - Constructeur de la classe, qui initialise le graphe (tg), le départ (depart) et l'ordre de visite (l_ordre_visite).
2. **randomListe(self)** :
    - Méthode pour générer une liste aléatoire de lieux visités, en considérant la capacité de vol.
3. **unVoisinListe(self)** :
    - Retourne un voisin sous forme de liste, en modifiant légèrement l'ordre des lieux visités.
4. **unVoisin(self)** :
    - Retourne une solution basée sur le voisinage généré par **unVoisinListe**.
5. **lesVoisins(self)** :
    - Génère une liste de voisins en tirant aléatoirement des chemins, divisés en un certain nombre de voisins.
6. **eval(self)** :
    - Évalue la solution courante en mesurant la distance totale parcourue à travers les lieux visités.
7. **nlleSolution(self)** :
    - Génère une nouvelle solution aléatoire à partir de la solution actuelle.
8. **displayPath(self)** :
    - Affiche le chemin parcouru par R2D2 sous forme d'un chemin dans le graphe.
9. **hash(self)** et **eq(self, o)** :
    - Méthodes pour hacher et comparer des instances de UneSolution.
10. **str(self)** :
    - Représentation sous forme de chaîne de caractères de la solution courante.

Détail des Méthodes

1. **init(self, tg: GrapheDeLieux, depart: int = 0, l_ordre_visite=None)** :
    - Initialise le graphe, le départ et l'ordre de visite. Si l_ordre_visite est fourni, il est copié ; sinon, un chemin aléatoire est généré.
2. **randomListe(self)** :
    - Cette méthode génère une liste de lieux en incluant le départ et les lieux restants sous forme aléatoire. Cela utilise une approche plus rapide en excluant les routes existantes.
3. **unVoisinListe(self)** :
    - Retourne un voisin sous forme de liste modifiée, où un sous-intervalle est inversé.
4. **lesVoisins(self)** :

- Génère une liste de voisins aléatoires en générant plusieurs chemins et en tirant un certain nombre de voisins, équilibrant exploration et rapidité.

5. **eval(self)** :
   - Calcule la distance totale parcourue entre les lieux visités dans l'ordre spécifié.
6. **nlleSolution(self)** :
   - Crée une nouvelle solution aléatoire en utilisant randomListe().

SolverTabou (Recherche Tabou)

Le **SolverTabou** est une métaheuristique de **recherche locale** avancée. Il fonctionne en explorant les solutions voisines d'une solution courante pour trouver des solutions meilleures, tout en utilisant une **mémoire** pour éviter de revisiter des solutions déjà explorées. Cette mémoire est représentée par une **liste tabou** qui contient les solutions récentes ou les mouvements récemment effectués. L'objectif est d'éviter que l'algorithme ne se retrouve coincé dans un **optimum local**.

Fonctionnement du SolverTabou :

1. **Exploration du voisinage** : Le SolverTabou explore les voisins d'une solution courante en effectuant des échanges de ville**s** dans le chemin, par exemple, en inversant l'ordre de deux villes dans la séquence de visites, ou comme je l'ai fait avec plusieurs villes d'un coup.
2. **Liste Tabou** : Lorsqu'une solution est explorée, elle est ajoutée à la liste tabou pour un nombre défini d'itérations. Cela empêche l'algorithme de revenir sur des solutions déjà testées récemment, réduisant ainsi le risque de se retrouver dans des cycles.
3. **Sortir des optima locaux** : La liste tabou permet au SolverTabou de s'échapper des minima locaux, en forçant parfois l'algorithme à accepter des mouvements qui ne sont pas immédiatement avantageux mais qui peuvent mener à une meilleure solution à long terme.

Avantages pour le TSP :

- **Efficacité pour l'optimisation** : Cette approche permet de trouver des solutions de haute qualité en explorant des voisins prometteurs tout en évitant de tourner en rond dans des optima locaux.
- **Flexibilité** : La taille et le contenu de la liste tabou peuvent être ajustés pour optimiser les performances selon la complexité du problème.

SolverHC (Hill Climbing)

Le **SolverHC**, ou **Hill Climbing**, est une métaheuristique de recherche locale plus simple et directe. Contrairement au SolverTabou, il ne garde pas de mémoire des solutions précédentes et explore les voisins d'une solution en fonction de la qualité immédiate des voisins.

Fonctionnement du SolverHC :

1. **Exploration du voisinage** : Comme dans le SolverTabou, le SolverHC explore les voisins d'une solution en échangeant des villes dans le chemin.
2. **Critère d'acceptation** : Le voisin est accepté uniquement si sa valeur (longueur du chemin) est meilleure que celle de la solution actuelle. Cela signifie que l'algorithme cherche continuellement à améliorer la solution courante.
3. **Risque de se bloquer** : Un inconvénient majeur du SolverHC est qu'il peut facilement se bloquer dans un minimum local. Si le voisinage immédiat d'une solution est meilleur, l'algorithme accepte la solution. Cependant, si l'algorithme arrive dans un minimum local sans meilleure solution dans les environs immédiats, il peut s'arrêter sans avoir trouvé la meilleure solution globale.

Avantages pour le TSP :

- **Simplicité et rapidité** : Le SolverHC est simple à mettre en œuvre et peut converger rapidement vers une solution acceptable.
- **Solution rapide dans de petites instances** : Pour des instances avec peu de villes, cette méthode peut être suffisante et relativement rapide pour trouver une solution convenable.

Comparaison entre SolverTabou et SolverHC :

En utilisant les deux métaheuristiques, nous pouvons comparer leur efficacité et leur comportement face à différentes instances du problème TSP. Voici quelques points de comparaison :

- **Robustesse** : Le SolverTabou est plus robuste car il dispose d'une mémoire et évite les pièges des optima locaux, alors que le SolverHC risque de rester coincé dans des minima locaux.
- **Complexité** : Le SolverTabou est plus complexe à implémenter et nécessite une gestion de la mémoire et de la liste tabou, ce qui le rend plus coûteux en temps de calcul que le SolverHC, qui est plus simple et rapide.
- **Convergence** : Le SolverHC peut être plus rapide pour les petites instances ou quand une solution acceptable suffit, mais le SolverTabou tend à fournir des solutions plus proches de l'optimum global, surtout pour des problèmes plus complexes.

Conclusion:

La vrai difficulté ici est de générer des voisins correctement, vous m’ avez expliqué en tp que plusieurs méthodes était possible j’ai choisi la méthode que vous m’aviez expliqué au tableau en coupant la liste de sommet d’une taille n et en inversant. L’algo de coloration et le HC se lancent très bien mais le dernier test avec le tabou prend un temps fou!

Remarque: Ma version 1 générait exactement TOUS les voisins en interchangeant 1 sommet à la fois. C'est théoriquement le plus juste mais mon ordi ne le supporte pas!
Il faut bien penser à renvoyer un singleton dans unVoisin() dû à votre implémentation du solveurHC.

IV - Etape 4

Etape 4 = TP 9 et 10 Réalisation de la 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 deja proposes.

Modélisation du problème

1. **Problème** : Coloration de graphe
2. **Encodage sous forme de graphe de contraintes** : Chaque nœud du graphe représente une zone à colorier, et les arêtes entre les nœuds symbolisent les relations d'adjacence entre les zones. La contrainte principale est que deux zones adjacentes ne doivent pas partager la même couleur.
3. **Algorithme utilisé** : **SolverCSP** (algorithme de résolution de problèmes de satisfaction de contraintes), conçu pour gérer les contraintes et trouver une solution qui respecte les conditions du problème.

V - Etape 5

Etape 5 = TP 11 et 12 Réalisation de la tache 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 tache 2, et cherche `a le résoudre en l'exprimant `à l'aide de formules mathématiques. Vous devrez proposer un encodage approprié en utilisant le langage ZIMPL et résoudre le problème en utilisant le solver SCIP.

Le programme permet de résoudre le problème du voyageur de commerce (TSP) en utilisant le solveur SCIP et l'encodage ZIMPL. Il commence par définir le problème en ZIMPL, où l'ensemble des villes et des arêtes entre elles est décrit, ainsi que les contraintes nécessaires pour garantir une solution valide.

Pour chaque taille de problème (de 6 à 19 villes), un fichier ZIMPL est généré en modifiant dynamiquement le nombre de villes. Ce fichier est ensuite utilisé par SCIP pour résoudre le problème, et le résultat, qui est la distance optimale (c'est-à-dire le coût du chemin le plus court), est extrait à partir des logs générés par le solveur.

Le modèle ZIMPL inclut des variables binaires représentant les arêtes entre les villes, une fonction objectif qui minimise la distance totale parcourue, ainsi que des contraintes pour garantir qu'il n'y a pas de sous-tours et que chaque ville est visitée une seule fois.

À la fin de chaque résolution, la valeur de l'objectif est extraite et affichée, ce qui permet de comparer les résultats pour différentes tailles de problèmes (nombre de villes).

PS: En vous remerciant pour l'aide par mail!