Journal de Bord

INTELLIGENCE ARTIFICIELLE, RESOLUTION DE PROBLEMES

LOUAHADJ Aniss
2A - SRI - UPSSITECH

Introduction

Ce document représente le journal de bord de mon avancé sur les 12 séances de travaux pratiques dans la matière « Intelligence artificielle, résolution de problèmes ». Vous trouverez, pour chaque séance datée, le compte rendu du travail effectué pendant ces dernières.

Séance du 26 Janvier 2021 : Etape 1

Lors de cette séance nous avons passé un long moment à installer l'IDE Eclipse ainsi que les différentes librairies imposées sur la page Moodle.

Nous nous sommes ensuite intéressés au vocabulaire à utiliser. Nous aurons un nombre de variable :

nbVariable = nombreDeCouleurs*nombreDeSommets

Pour représenter une variable nous avons un chiffre positif pour dire qu'elle est vraie et son négatif pour dire que celle-ci est fausse.

Exemple : Le sommet « a » est rouge aura pour variable « Ra » qui sera codée « 1 ».
La variable « non Ra » sera « -1 ».
Si on a 200 sommets, nous aurons des variables de -200 à 200.

Nous avons également lu la documentation de différentes classes et essayé de comprendre ce qu'est censé représenter la variable « base ».

Séance du 02 Février 2021 : Etape 1

La première séance nous a permis de bien assimilé le sujet et plus particulièrement la tâche 1.

Pour répondre à celle-ci et compléter le fichier Etape1.java, nous avons utilisé les équations logiques que nous avons eu la chance de trouver pour un problème similaire en travaux dirigés pour une coloration à trois couleurs. Cependant, notre code est un code générique adapté à un nombre de couleurs quelconque.

Contrainte 1 : Un sommet doit avoir une couleur.

Contrainte 2 : Un sommet ne peut avoir qu'une couleur.

Contrainte 3 : Deux sommets adjacents ne peuvent avoir la même couleur.

Notre classe etape1 possède trois attributs : Un graphe de lieux, une liste de liste représentant l'ensemble des contraintes ainsi que notre vocabulaire, et enfin un entier nbVariables calculé lors de l'appel à la fonction majBase.

Notre constructeur ne prend en paramètres que ceux utiles à la classe GrapheDeLieux et instancie ainsi l'attribut correspondant. Il crée également une liste de liste vide et l'attribue à base.

La fonction **majBase** est la fonction où la grande majorité du travail est effectuée. Celle-ci crée notre vocabulaire en fonction du nombre de couleurs passées en paramètres. Par la suite, en fonction du nombre de sommets présents dans notre graphe de lieux, la variable base est mise à jour suivant les trois contraintes que l'on a codé. Nous n'avons pas eu besoin de crée une variable contenant notre vocabulaire car nous avons remarqué qu'en codant la première contrainte, le vocabulaire y était retrouvé. Les contraintes 2 et 3 font donc appel aux éléments créés pas la première contrainte.

La fonction **execSolver()** fait appel au SolverSAT jugé le plus approprié pour notre problème.

La fonction **affBase()** se contente d'afficher la variable base.

Séance du 05 Février 2021 : Etape 2

Lors de cette séance nous avons entreprit la réalisation de l'étape 2 de ce projet, autrement dit la réalisation de la tâche 2 pour 3 cas distincts.

Nous commençons par le **cas 1**. Ici, nous devons compléter le code de la classe EtatCas1 sensée être la représentation d'un état afin que le solveur puisse choisir le plus court chemin entre deux lieux du monde. Notre problème étant relativement simple, nous avons décidé qu'un état sera représenté par le lieu de départ (un entier symbolisant le sommet où le robot se situe), le lieu d'arrivée (un entier symbolisant le sommet final/la destination) vers lequel R2D2 doit se diriger ainsi que le graphe du monde dans lequel le robot évolue. Nous réalisons ainsi trois constructeurs différents.

- **Constructeur 1** – Paramètres : ***Graphe de lieux*** :
  Nous passons en paramètres de ce constructeur uniquement un graphe de lieux. Les lieux de départ et d'arrivée de l'état instancié seront respectivement par défaut le premier et le dernier sommet présents dans le graphe.
- **Constructeur 2** – Paramètres : ***Graphe de lieux et lieu de départ*** :
  Nous passons en paramètres de ce constructeur le graphe de lieux ainsi que le lieu de départ. Le lieu d'arrivée de l'état instancié sera par défaut le dernier état présent dans le graphe.
- **Constructeur 3** – Paramètres : ***Graphe de lieux, lieu de départ et lieu d'arrivée*** :
  Nous passons en paramètres de ce constructeur le graphe de lieux, le lieu de départ ainsi que le lieu d'arrivée de l'état instancié.

Nous sommes ensuite passé à la complétion du code des méthodes présentes dans le fichier :

-Fonction **estSolution()** : fonction renvoyant un booléen disant si l'état courant correspond à l'état d'arrivée prévu.

-Fonction **successeurs()** : fonction renvoyant une liste contenant l'ensemble des états successeurs de notre état courant, autrement dit les états dont le lieu de départ

est un sommet adjacent (puisqu'il nous travaillons sur des graphes non orientés) à l'état de départ de notre état courant.

-Fonction **h()** : fonction permettant de récupérer l'heuristique de l'état courant. Pour se faire, nous utilisant simplement la fonction calculant la distance euclidienne entre de lieux déjà présente dans la classe GrapheDeLieux.

-Fonction **k()** : fonction permettant de récupérer le coût du passage entre un état donné et l'état courant. Là encore nous utilisons une fonction présente dans la classe GrapheDeLieux.

La séance étant terminé nous n'avons pas eu le temps d'aller plus loin.

Séance du 09 Février 2021 : Etape 2

Lors de cette séance nous avons continué notre code de la séance d'avant.

-Fonction **displayPath()** : fonction affichant le chemin allant de l'état initial à l'état final en suivant l'ordre de passage.

-Fonction **compareTo()** : fonction comparant l'état courant à l'état passé en paramètre. Dans cette fonction nous nous assurons tout d'abord que l'objet passé en paramètre est bien de la classe EtatCas1. Ensuite, nous faisons simplement la différence entre le lieu de départ de l'état courant et le lieu de départ de l'état passé en paramètre.

-Fonction **hashCode()** : fonction permettant de récupérer le code de hachage de l'état courant. Nous avons décidé de définir celui-ci comme étant la somme du nombre représentant le lieu de départ, le nombre représentant le lieu d'arrivée ainsi que le nombre de sommets du graphe. Cependant, dans notre cas, ce code aurait tout simplement pu être le nombre représentant le lieu de départ.

-Fonction **equals()** : fonction permettant de comparer si l'état courant est égal à l'état passé en paramètres. Pour se faire, nous vérifiant tout d'abord si l'objet passé en paramètres et de la classe EtatCas1. Si non, la fonction renvoie false. Si l'objet est bien une instance de EtatCas1, la fonction renvoie un booléen comparant l'égalité entre les hasCode des deux objets.

-Fonction **toString()** : fonction renvoyant une chaine de caractères afin de symboliser l'état courant pour un affichage.

Après avoir fini le code de notre classe EtatCas1, nous avons testé cette dernière à l'aide de la classe Etape2. Pour se faire, nous avons décidé d'utiliser un solver AStar car nous travaillons sur les heuristiques et le poids des arcs.

Suite au test nous avons des résultats satisfaisants.

Séance du 12 Février 2021 : Etape 2

Lors de cette séance nous avons entreprit la réalisation du cas 2 de l'étape 2. Nous devons donc compléter le code de ce dernier toujours censé être la représentation d'un état. Cependant, cette fois-ci, le solveur doit chercher le chemin le plus court permettant de passer dans chaque lieu une seule fois puis de revenir au point de départ.

Afin de représenter chaque état, nous aurons besoin d'un graphe de lieux, d'un lieu de départ-arrivée ainsi que d'une liste de l'ensemble des états visités. Cette liste est censée posséder un exemplaire unique de chaque sommet à l'exception du sommet de départ

- **Constructeur 1** – Paramètres : **Graphe de lieux** :
  Nous passons en paramètres de ce constructeur uniquement un graphe de lieux. Le lieu de départ/arrivée sera le sommet 0 par défaut. Nous attribuons une liste vide à la liste des états vus et nous ajoutons ce premier sommet.
- **Constructeur 2** – Paramètres : **Graphe de lieux et lieu de départ** :
  Nous passons en paramètres de ce constructeur le graphe de lieux ainsi que le lieu de départ. Nous créons une liste vide et ajoutons ce premier lieu à celle-ci.
- **Constructeur 3** – Paramètres : **Etat** :
  Il s'agit d'un constructeur par copie. Il permet donc de créer un état possédant les mêmes valeurs d'attributs que l'état passé en paramètres.

Nous sommes ensuite passé à la complétion du code des méthodes présentes dans le fichier :

-Fonction **estSolution()** : fonction renvoyant un booléen disant si l'état courant correspond à l'état d'arrivée prévu (donc s'il s'agit du même sommet) et si la liste des états vus a été convenablement remplie (si le nombre de sommets qu'elle contient est bien égal au nombre de sommets du graphe de lieux + 1).

-Fonction **successeurs()** : fonction renvoyant une liste contenant l'ensemble des états successeurs de notre état courant, autrement dit les états dont le lieu de départ/arrivée est un sommet adjacent (puisqu'il nous travaillons sur des graphes non orientés) à l'état de départ de notre état courant mais qui ne sont pas déjà présents dans la liste des états vus de l'état courant.

-Fonction **h()** : fonction permettant de récupérer l'heuristique de l'état courant. Pour se faire, nous faisons la soustraction du nombre de sommets présents dans notre graphe de lieux par la taille de la liste des états que nous avons visité et nous multiplions le tout par une valeur importante.

-Fonction **k()** : fonction permettant de récupérer le coût du passage entre un état donné et l'état courant. Pour se faire, nous utilisons simplement la fonction permettant de renvoyer le coup du passage entre deux sommets du graphe. Les sommets étant les derniers visités de chaque état.

-Fonction **displayPath()** : fonction affichant le cycle passant par tous sommets en utilisant une map contenant les sommets visités et l'état par lequel nous sommes passés pour y arriver.

-Fonction **compareTo()** : fonction comparant l'état courant à l'état passé en paramètre. Dans cette fonction nous nous assurons tout d'abord que l'objet passé en paramètre est bien de la classe EtatCas2. Ensuite, nous comparons simplement chaque attribut des deux états.

-Fonction **hashCode()** : fonction permettant de récupérer le code de hachage de l'état courant. Nous avons décidé de définir celui-ci comme étant la somme de l'entier représentant l'état de départ/arrivée, le hashCode de la liste des états vus ainsi que le nombre de sommets présents dans le graphe.

-Fonction **equals()** : fonction permettant de comparer si l'état courant est égal à l'état passé en paramètres. Pour se faire, nous vérifiant tout d'abord si l'objet passé en paramètres et de la classe EtatCas2. Si non, la fonction renvoie false. Si l'objet est bien une instance de EtatCas2, la fonction renvoie un booléen comparant l'égalité entre les hasCode des deux objets.

-Fonction **toString()** : fonction renvoyant une chaine de caractères afin de symboliser l'état courant pour un affichage.

Nous ajoutons également des **getters** et des **setters** que nous avons utilisé dans le corps des fonctions mentionnées ci-dessus.

Après avoir fini le code de notre classe EtatCas2, nous avons testé cette dernière à l'aide de la classe Etape2. Pour se faire, nous utilisons encore une fois un solveur aStar.

Suite au test nous avons des résultats satisfaisants.

Séance du 16 Février 2021 : Etape 2

Lors de cette séance nous avons entreprit la réalisation du cas 3 de l'étape 2. Nous devons donc compléter le code de ce dernier toujours censé être la représentation d'un état et répondant à la même problématique que le cas 2 sauf que cette fois-ci il n'est plus nécessaire de prendre en compte les arcs imposés entre deux sommets du graphe.

Afin de représenter chaque état, nous aurons besoin d'un graphe de lieux, d'un lieu de départ-arrivée ainsi que d'une liste de l'ensemble des états visités. Cette liste est identique à celle vu lors de la séance précédente.

- **Constructeur 1** – Paramètres : **Graphe de lieux** :
  Nous passons en paramètres de ce constructeur uniquement un graphe de lieux. Le lieu de départ/arrivée sera le sommet 0 par défaut. Nous attribuons une liste vide à la liste des états vus et nous ajoutons ce premier sommet.
- **Constructeur 2** – Paramètres : **Graphe de lieux et lieu de départ** :
  Nous passons en paramètres de ce constructeur le graphe de lieux ainsi que le lieu de départ. Nous créons une liste vide et ajoutons ce premier lieu à celle-ci.
- **Constructeur 3** – Paramètres : **Etat** :
  Il s'agit d'un constructeur par copie. Il permet donc de créer un état possédant les mêmes valeurs d'attributs que l'état passé en paramètres.

Nous sommes ensuite passé à la complétion du code des méthodes présentes dans le fichier :

-Fonction **estSolution()** : fonction renvoyant un booléen disant si l'état courant correspond à l'état d'arrivée prévu (donc s'il s'agit du même sommet) et si la liste des états vus a été convenablement remplie (si le nombre de sommets qu'elle contient est bien égal au nombre de sommets du graphe de lieux + 1).

-Fonction **successeurs()** : fonction renvoyant une liste contenant l'ensemble des états successeurs de notre état courant, autrement dit les états dont le lieu de départ/arrivée n'est pas encore présent dans la liste des états vus.

-Fonction **h()** : fonction permettant de récupérer l'heuristique de l'état courant. Pour se faire, nous faisons la soustraction du nombre de sommets présents dans notre graphe de lieux par la taille de la liste des états que nous avons visité et nous multiplions le tout par le coup minimum entre deux sommets du graphe par les airs.

-Fonction **k()** : fonction permettant de récupérer le coût du passage entre un état donné et l'état courant. Pour se faire, nous renvoyons simplement la distance entre le dernier sommet vu par l'état courant et le dernier sommet vu par l'état passé en paramètres.

-Fonction **displayPath()** : fonction affichant le cycle passant par tous sommets en utilisant une map contenant les sommets visités et l'état par lequel nous sommes passés pour y arriver.

-Fonction **compareTo()** : fonction comparant l'état courant à l'état passé en paramètre. Dans cette fonction nous nous assurons tout d'abord que l'objet passé en paramètre est bien de la classe EtatCas3. Ensuite, nous comparons simplement chaque attribut des deux états.

-Fonction **hashCode()** : fonction permettant de récupérer le code de hachage de l'état courant. Nous avons décidé de définir celui-ci comme étant la somme de l'entier représentant l'état de départ/arrivée, le hashCode de la liste des états vus ainsi que le nombre de sommets présents dans le graphe.

-Fonction **equals()** : fonction permettant de comparer si l'état courant est égal à l'état passé en paramètres. Pour se faire, nous vérifiant tout d'abord si l'objet passé en paramètres et de la classe EtatCas3. Si non, la fonction renvoie false. Si l'objet est bien une instance de EtatCas3, la fonction renvoie un booléen comparant l'égalité entre les hasCode des deux objets.

-Fonction **toString()** : fonction renvoyant une chaine de caractères afin de symboliser l'état courant pour un affichage.

Nous ajoutons également des **getters** que nous avons utilisé dans le corps des fonctions mentionnées ci-dessus.

Après avoir fini le code de notre classe EtatCas3, nous avons testé cette dernière à l'aide de la classe Etape2. Pour se faire, nous utilisons encore une fois un solveur aStar.

Suite au test nous avons des résultats satisfaisants.

Séance du 17 Février 2021 : Etape 3

Lors de cette séance, nous voulons comparer deux solveurs différents. Le solveur hilClimbing ainsi que le solveur tabou. Nous considérons encore les éléments fournis lors de la séance précédentes (nous considérons que le robot peut voler et faisons donc abstraction des arcs entre les sommets du graph). Afin d'utiliser les solveurs proposés nous devons donc compléter la classe UneSolution censée représenter une solution acceptable.

Afin de représenter solution, nous aurons besoin d'un graphe de lieux ainsi que d'une liste d'entiers représentant une suite de sommets symbolisant une solution.

- **Constructeur 1** – Paramètres : **Graphe de lieux** :
  Nous passons en paramètres de ce constructeur uniquement un graphe de lieux. La liste des sommets représentant une solution sera par défaut les sommets dans l'ordre présent dans le graphe (de 0 au nombre de sommets-1).
- **Constructeur 2** – Paramètres : **Graphe de lieux et liste d'entiers** :
  Nous passons en paramètres de ce constructeur le graphe de lieux ainsi qu'une suite d'entiers représentant une solution possible par laquelle nous souhaiterions commencer.

Nous sommes ensuite passé à la complétion du code des méthodes présentes dans le fichier :

-Fonction **lesVoisins()** : fonction renvoyant une liste contenant les voisins possibles à partir de la solution courante. Un voisin de notre solution courante sera une solution dont la liste des sommets sera identique à la solution courante à l'exception près que deux sommets se suivants seront intervertis.
Exemple : Une solution ayant pour liste d'entiers 0-1-2-3 aura pour voisins 1-0-2-3, 0-2-1-3, 0-1-3-2 et 3-1-2-0.

-Fonction **unVoisin()** : fonction renvoyant un voisin aléatoire présent dans la liste des voisins possibles de l'état courant.

-Fonction **eval()** : fonction renvoyant la valeur de la solution courante. Cette valeur est la somme des distances euclidiennes des sommets qui se suivent dans la solution courante jusqu'à revenir au premier sommet.

-Fonction **nelleSolution ()** : fonction renvoyant une solution avec un ordre aléatoire des sommets.
Ici encore nous codons les fonctions **displayPath**, **hashCode**, **equals** et **toString** qui ont le même rôle que précédemment.

Nous avons ensuite testé notre travail en utilisant deux solveurs pour chacun des cas proposés dans l'étape 3.

Avec le hill climbing, nous n'avons aucune certitude sur le fait d'obtenir la meilleure solution. Si l'algorithme trouve un minimum local il s'arrêtera au lieu de revenir en arrière ou de poursuivre et tester des solutions qui apparaissent plus loin. A partir d'une solution donnée, il cherchera une meilleure solution parmi la liste des voisins de la solution courante. S'il en trouve une il répètera l'opération. S'il n'en trouve pas il considérera que la solution courante est la meilleure.
Le solveur tabou lui accepte des solutions non améliorantes dans sa méthode de résolution tout en gardant en mémoire des solutions déjà visitées afin d'éviter de boucler. Il choisi la meilleure solution dans le voisinage de la solution courante même si celle-ci n'est pas améliorante.

En effectuant nos tests, nous pouvons observer que le tabou est largement supérieur au hill climbing. La différence est difficilement observable pour un faible nombre d'états. Cependant, très vite la supériorité du tabou se fait sentir pour un nombre d'états plus élevé et pour un nombre d'essais plus important.

Bien que la solution obtenue a des chances de ne pas être la solution la plus optimale, le temps de calcul est beaucoup plus rapide que celui de l'Astar et la solution obtenue est largement acceptable.

Séance du 03 Mars 2021 : Etape 4

Lors de cette séance nous devions simplement manipuler le solveur CSP fourni. Ce dernier prend en paramètres un graphe de lieux ainsi que le nombre de couleurs que l'on souhaiterait utiliser pour effectuer une coloration sur ce dernier. Nous avons ainsi N variables représentant le nombre de sommets ainsi que M couleurs.
Chaque variable possède un domaine de définition étant [0,1, 2, …, M-1]. Ces éléments représentent les variables associées à leur encodage comme vu lors de l'étape 1.
Ces variables associées à leur encodage ont ensuite des contraintes identiques à celles que nous avons définies lors de la première étape.

Contrainte 1 : Un sommet doit avoir une couleur.

Contrainte 2 : Un sommet ne peut avoir qu'une couleur.

Contrainte 3 : Deux sommets adjacents ne peuvent avoir la même couleur.

Le solveur CSP utilise un algorithme de backtrack. Lorsqu'il associe un encodage à une variable il vérifie la vraisemblance à l'aide des contraintes définies et passe à la variable suivante. Si un encodage ne répond pas à l'une des contraintes, l'algorithme revient sur sa décision et associe un nouvel encodage tel que défini par son domaine de définition précédemment. Si aucun encodage n'est possible pour une des variables, le solveur nous renverra un NOK. Si toutes les variables ont pu être encodées sans aller à l'encontre d'une des contraintes, alors il est possible de faire une coloration du graphe avec le nombre de couleurs passées en paramètres et le solveur renverra OK.

Séance du 10 Mars 2021 : Etape 5

Lors de cette séance, nous devions à nouveau résoudre le problème de la tâche 2. Nous devons écrire sous une forme mathématique les contraintes que nous avons choisies déjà défini.
Nous considérons donc un graphe de N sommets composé de $\frac{N(N-1)}{2}$ arêtes. Nous aurons ainsi autant de variables binaires constituant un cycle avec tous les sommets du graphe. Une variable à 1 signifie qu'elle appartient au cycle et que donc que les deux sommets qu'elle est sensée représentée sont reliés dans cette solution. Si elle est à 0 cela veut dire qu'elle n'y appartient pas.

Ce problème étant beaucoup plus dur que les précédents et ayant eu beaucoup de mal nous avons simplement trouvé une solution sur internet que nous avons tenté de comprendre et modifier.

Afin de résoudre les différents problèmes de cette étape il faut lancer le solveur SCIP sur le fichier zpl que nous avons créé en fonction des différentes villes.