Projet final d'Intelligence Artificielle

Alexis HUMMEL, Izabell ISKANDAR, Salim ZERHOUNI

Analyse du problème et de sa résolution

La définition du problème :

Pour modéliser un état, nous avons créé la classe **City** qui a comme attributs x et y qui représentent les coordonnées de la ville ainsi qu'une liste statique listCity qui correspond à la liste de toutes les villes données.
Nous avons ensuite créé la classe **Visit** qui représente un état avec comme champs la ville courante dans laquelle nous nous trouvons ainsi que la liste des villes qui ne sont pas encore visitées.
Soit C0 la ville initiale et C1,...Cn les autres villes
Soit 0<=i<=n
État initial : Ville courante initiale et liste de villes non visitées dont les élements sont toutes les villes de listCity : {C0,[C0,C1,…,Cn]}
État final : Ville courante initiale et liste de villes non visitées vide : {C0,[]}
La méthode *isSolved()* permet de vérifier que nous nous trouvons bien à l'état final.

Pour modéliser une action, nous implémentons la méthode *goTo()* qui permet d'aller d'une ville à une autre, mise en argument.
ACTION(s) : Aller dans une ville non visitée et différente de la ville courante de s, donnée en argument dans *goTo() : (--> Ci) avec Ci dans les villes non visitées. Si i = 0, il faut que seule C0 soit non visitée.*

RESULT(s,a) : Après avoir appelé *goTo()* à l'état s, avec comme paramètre la ville Ci, on se retrouve à l'état s' dans lequel on est en ville courante Ci qu'on supprime de la liste des villes non visitées : RESULT(*{Ci,[Cj,...,Ck]}, (--> Cm)) = {Cm,[Cj,…,Ck]\{Cm}}.*
c(s,a,s') = distance entre la ville courante de s et celle de s', pour cela, nous utilisons *distance()* calculant la distance euclidienne entre la ville courante de l'état, qui appelle la méthode, et la ville courante de l'état en paramètre.

Nous avons implémenté une classe **State** que nous allons détailler plus tard. Un objet de telle classe comporte un champ **Visit** décrivant l'état actuel, l'état parent (comportant la ville courante avant la dernière visite) et une heuristique **h(n)**. Ce nombre est calculé grâce à notre méthode *heuristique()* de la classe **Visit**, prenant la distance entre la ville courante et la ville initiale dans un arbre couvrant minimal passant par les autres villes. Celle-ci est admissible car la distance entre deux sommets dans un arbre couvrant minimal ne surestime jamais la distance minimale pour y parvenir. Elle est en plus valable car pour chaque état, son heuristique sera plus petite que la somme de l'heuristique de son suivant et il en va de même à l'étape suivante. Nous avons également un champ costAction qui calcule la fonction coût **g(n)**. Celle-ci correspond simplement à c(s0, a, s1) + … + c(si-1, a, si) avec c(si-1, a, si) = distance entre ville courante de si-1 et ville courante de si.

Description du code

Recherche informée

La classe **State** est la classe principale dans le projet pour la partie de la recherche informée. Nous avons déjà détaillé les différents champs plus haut. L'attribut statique *nbVisit* permet de créer à chaque appel à la classe l'attribut *id* unique qui représente un identifiant unique de l'état State en question. La méthode *expand()* renvoie une liste de State (états ou encore nœuds) fils (=enfants) de l'état sur lequelle la méthode s'applique. La méthode *evaluate()* calcule g(n) + h(n) de la ville en question. La classe **Generation** est une classe permettant de générer des listes de n villes aléatoirement grâce à la méthode *randomGeneration().*
La classe **AStar** implémente l'algorithme d'A* dans la méthode *algo_AStar()*. Comme vu dans le cours, nous initialisons deux listes frontier et explore (plus précisément cette dernière est un stack). Nous initialisons explore à un seul état, l'état initial. Si nous explorons tout sans trouver de solution, nous renvoyons une erreur. Sinon, nous continuons jusqu'à trouver la solution, qui sera optimale. Pour cela, à chaque itération, nous prenons l'élément en tête de stack, appliquons *expand()* pour prendre tous ses successeurs (sauf le parent) que nous mettons dans frontier. Puis nous prenons le minimum des états du frontier en prenant en compte les f(n) = g(n) + h(n) grâce à la méthode *evaluate().* Nous l'avons fait grâce à un comparateur écrit dans **State** pour pouvoir utiliser la méthode *min()* de la classe **Collections** afin de trouver facilement le minimum. Nous ajoutons cet état minimum dans le stack explored et l'enlevons de frontier. Si la solution est trouvée (grâce à *isSolved()*), nous nous arrêtons et remontons parent par parent pour retrouver le chemin complet, sinon, nous continuons à la prochaine itération.

Recherche locale

Pour la recherche locale nous avons créer 2 classes en plus.
La classe **Hamilton** représente un cycle hamiltonien : on utilise une ville de départ et une liste des autres villes (générée par défaut lors de la création du tout premier cycle). Cette liste définit le cycle, par exemple si on a A comme ville de départ et L = [B,C,D] alors le cycle sera A, B, C, D pour ensuite retourner sur A.
On a également la classe **hamNeighboor.** Cette dernière va générer l'ensemble des voisinages possibles pour un état (un cycle donc), i.e l'ensemble de possibilités lorsqu'on applique 2-opt pour un cycle donné. Cette classe est dotée de la fonction *hillClim()* qui va chercher le meilleur voisin pour un cycle. Une fois trouvé, on applique à nouveau *hillClim()* sur le cycle obtenu jusqu'à arriver sur un optimum (un cycle qui n'a pas de meilleurs voisins). On est ici dans le cas d'une fonction à minimiser, à savoir le poids d'un cycle.

Pour l'Algorithme *algo_LocalBeamSearch*(int k), nous avons créer une nouvelle classe appelée **LocalBeamSearch** qui contient ce dernier. Dans cet algorithme on utilise les mêmes idées d'état et de voisinage que pour l'algorithme A*, contrairement à l'algorithme HillClimbing. *algo_LocalBeamSearch(int k)* prend en paramètre un nombre k qui va représenter le nombre de nœuds (états) qui vont être choisi pour la recherche local. À chaque itération de la boucle

while on prends les voisins des nœuds explorés, on les trie en ordre croissant en fonction des évaluations calculés grâce à la méthode *evaluate()* qui somme l'heuristique et le coût de l'action. Nous utilisons le même comparateur que pour A* afin d'utiliser la méthode *sort()* de la classe **Collections**. On choisit les k premiers meilleurs états voisins que l'on rajoute dans la liste des états explorés et on vérifie si l'un d'eux est l'état final, si oui on sort de la boucle while, sinon on continue la recherche. Après la boucle while on retrouve le chemin vers la racine afin de retourner le chemin solution.

Afin de mieux expliquer ce que l'on fait dans chaque algorithme, nous avons rajouté des commentaires dans le code résument chaque étape.

Exécution du code

Il suffit juste de suivre le README.md dans le fichier AI.

Description des testes

Vous pouvez retrouver la description des tests dans le fichier main.java du dossier AI/src.

Comparaisons des résultats

Voici les résultats que nous avons eu en faisant 3 tests avec la coordonnée d'abscisse maximale égale à 20, la coordonnée d'ordonnée maximale égale à 20, le nombre de villes à générer égale à 10, le nombre d'états à considérer pour l'Algorithme LocalBeamSearch égale à 5.

```
Temps d'exécution moyen de A* : 113.0 ms
Temps d'exécution moyen de HillClim : 2.3333333333333335 ms
Temps d'exécution moyen de LocalBeamSearch : 4.0 ms
Taux d'améliration moyen entre la solution initiale et la solution finale de HillClim : 0.37809036686583664
Taux d'améliration moyen entre la solution de Astar et la solution de HillClim : 0.07166090873450119
Taux d'écart moyen entre la solution de Astar et la solution de LocalBeamSearch : 0.2726425453419817
Coût total moyen pour l'algorithme Astar : 61.460607323247764
Coût total moyen pour l'algorithme HillClim : 67.06591431875752
Coût total moyen pour l'algorithme LocalBeamSearch : 84.65521751892908
```

Comme on peut le voir sur l'image au-dessus l'algorithme A* est le plus lent, ceci était prévisible étant donné qu'il cherche l'état avec la somme de l'heuristique et le coût d'action minimale parmi tous les états voisins. Plus il y a de ville, plus la taille de la frontier sera grande, et plus la recherche sera longue.
L'algorithme Hill Climbing est le plus rapide puisqu'il génère les états (chemins qui amènent vers l'état final) voisins, choisit le meilleur parmi (i.e. avec la plus évaluations) et s'arrête dès qu'il ne trouve plus de voisin meilleur. Il est possible que cet algorithme ne trouve pas la solution optimale.
L'algorithme LocalBeamSearch est également bien plus rapide que l'algorithme A* puisqu'il explore uniquement les k meilleurs voisins. Tout comme Hill Climbing, cet algorithme peut aussi ne pas trouver la solution optimale.

Vous pouvez également retrouver le taux d'amélioration moyen (sur les 3 tests) entre la solution initiale et la solution finale de HillClim, ainsi que le taux d'amélioration moyen entre la solution de Astar et la solution de HillClim et le taux d'écart moyen entre la solution de Astar et la solution de LocalBeamSearch. On remarque bien que l'algorithme Hill Climbing est plus proche de la solution optimale que l'algorithme LocalBeamSearch.

Il est également possible que l'algorithme LocalBeamSearch soit plus rapide, plus proche de la solution optimale et avec un meilleur coût total que l'algorithme Hill Climbing. Ceci est complètement aléatoire, dépend des villes de départ et des voisins générés à chaque fois.