Rapport

Sommaire :

Préambule :

Dans ce rapport, je vais vous expliquer les principales classes et méthodes de mon projet, tout en détaillant l'analyse et la résolution du problème. Les notations de cours d'IA seront utilisées pour expliquer la conception des algorithmes.

Pour rappel, le but de mon projet est de déterminer le plus court chemin Hamiltonien dans un ensemble de ville, le chemin reboucle sur la ville de départ. J'ai choisi pour résoudre ce problème l'algorithme A* pour la recherche informée et Hill-Climbing pour la recherche locale.

I) A*

Pour coder cet algorithme il fallait définir la notion d'état, pour cela j'ai créé une classe « State » qui va contenir un attribut de la classe ville (une ville est définie par ses coordonnées x, y et son nom), un autre attribut contient la liste des villes non visités, et bien sur un attribut père de type « State » pour connaitre le state parent grâce à un pointeur. Chaque état possède son cout de distance qui correspond à la somme des distances de l'état courant avec tous ses parents de la même lignée. Je vais maintenant présenter les fonctions principales de cette classe ***State*** ainsi que leur fonction :

- '*expand' :* cette méthode étant un état donné retourne tous les état voisin dans une liste. Pour cela on itère sur toutes les villes non visitées depuis l'état actuelle et chacune des villes non visitées va appartenir à un nouvel état(voisin) puis stocké dans la liste. La complexité de cette méthode est en O(n).

- '*heuristic*': cette méthode permet de calculer l'arbre couvrant minimal de notre état grâce à l'algorithme de Prim. La complexité de cette méthode est en $O(n^3)$.

- '*Goto'* : cette méthode permet de mettre à jour notre état en passant de la ville courante à la ville passé en paramètre, et met à jour état parent ainsi que sa distance.

Dans la classe principale ***Astar***, elle possède comme paramètre l'état initiale, un attribut explored de type liste qui va stocker les différents états visités pendant l'exécution de l'algorithme, un attribut frontier qui stocke les différents états non visités durant l'exécution. La méthode '*f_Astar'* débute par un état initial et choisi l'état avec la plus petite somme d'heuristique et distance, qu'on nomme par *min_no_explored*, on rajoute celui-ci dans explored et l'enlève de frontier, et on ajoute également dans frontier tous ses états voisins. La condition d'arrêt de notre algorithme est de reboucler sur la ville initiale. La méthode '*path_and_cout'* permet de retrouver le meilleur chemin dans l'attribut explored. Pour cela nous prenons la dernière ville explorée et prend son parent grâce au pointeur puis encore et encore son parent jusqu'à arriver sur null (état initiale) ainsi on trouve le nôtre meilleur chemin et en calcule au fur et à mesure son cout de distance.

L'algorithme Astar permet de trouver le meilleur chemin hamiltonien sur au plus 40 villes, au-dessus de ce nombre le temps de compilation est très grand.

II) Hill Climbing

Dans cet algorithme nous avons besoin également de définir la notion d'état. Dans la classe ***State_circuit,*** j'ai défini un état comme étant une liste de ville et ses voisins dans une liste de liste de ville. Les principales méthodes de cette classe sont les suivantes :

- '*find_voisins*' : cette méthode permet de retrouver tous les états voisins de l'état actuelle . Pour cela , il suffit juste d'inter-changer deux villes de l'état actuelle pour trouver un état voisins. Cette méthode rajoute toutes les possibilités d'inter-changement de deux villes. La complexité de cette méthode est en $O(n^2)$.

- '*cou_d'* : cette méthode renvoi le cout en distance de notre état , pour cela elle calcule la somme des distance entre les villes de notre liste .

- 'voisin_Min' : renvoi l'état voisin avec la plus petite distance grâce à 'cou_d'.

Dans la classe ***HillClimbing*** , se trouve notre algorithme principal fHC . Cet algorithme part d'un état initial et se déplace vers un état voisin avec la plus petite distance. L'algorithme stoppe quand l'état courant possède un meilleur cout en distance que tous ses voisins. Cette méthode peut trouver le meilleur chemin local avec au plus 200 villes. Le temps de compilation devient trop grand avec un plus grand nombre.

III) Tester le programme

Pour tester le programme il suffit juste d'exécuter le programme Main, puis une série de questions permettra d'initialiser aléatoirement le nombre de villes choisis et l'algorithme souhaité pour résoudre le problème. Le programme permet aussi d'afficher des images contenant le chemin de l'algorithme.

IV) Conclusion

Pour conclure , ce projet m'a permis de bien comprendre les rouages des algorithme de recherche locale et globale.