CPE Lyon - 4ICS - 2021/2022

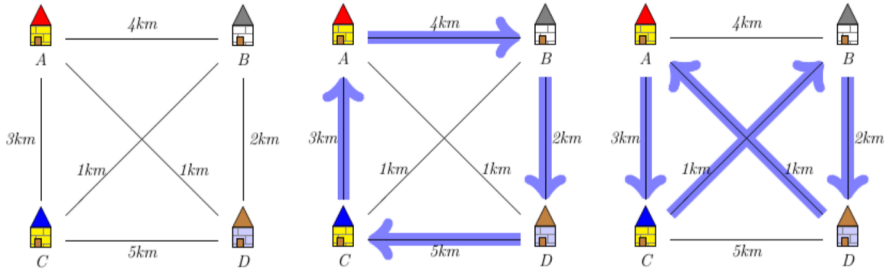
Structures de données et algorithmes avancés

Séance 12 - Algorithmes génétiques

Dans ce TP, vous allez coder un algorithme génétique pour trouver une solution approchée au **problème du voyageur de commerce** (*Traveling Salesman Problem* en anglais, ou TSP).

1 Enoncé du problème

Le problème est le suivant : un voyageur de commerce doit effectuer une tournée passant par n villes. Il peut les parcourir dans n'importe quel ordre (on considère que les villes sont reliées deux à deux), mais il ne doit passer qu'une seule fois par chaque ville et revenir à son point de départ et évidemment, il souhaite trouver le trajet le plus court (ou celui demandant le moins de temps). La figure ci-dessous représente une instance à 4 villes du problème, une solution à 14 km et la solution optimale à 7 km :



(a) Une instance à 4 villes (b) Solution naïve (c) Solution optimale

FIGURE 1 – Une instance du problème du voyageur de commerce (Wikipédia)

Ce problème, dont les premières mentions datent d'au moins 1832, est l'un des problèmes les plus célèbres en mathématiques et en informatique. En effet, il s'agit d'un de ces fameux problèmes dits "NP-complets", pour lesquels on ne connaît pas d'algorithme *efficace* donnant la solution optimale. Pour de grandes instances, on devra donc se contenter de *solutions approchées*, car on se retrouve face à une **explosion combinatoire** (par exemple, avec 71 villes, le nombre de chemins possibles est environ égal au nombre d'atomes dans l'univers connu !).

En raison de l'importance pratique de ce problème, dans des domaines aussi variés que le transport routier, la poste, la livraison de repas, l'inspection d'installations, l'astronomie (avec la minimisation des mouvements des grands téléscopes, très lents), l'industrie (minimiser le temps total mis par une fraiseuse à commande numérique pour percer n points dans une plaque de tôle), la biologie (séquençage du génôme)... il a été intensément étudié, et il existe de nombreuses heuristiques et métaheuristiques permettant d'obtenir des solutions approchées avec de bonne qualité. L'une de ces méthodes consiste à utiliser un algorithme génétique.

Question 1. Comment peut-on formuler ce problème en termes de graphes ?

Question 2. Quel est le nombre de trajets possibles pour une instance à n villes ?

2 Résolution à l'aide d'un algorithme génétique

Question 3. Décrivez ce que sont les *individus* dans le cas présent. Comment les représenter par un *chromosome* ?

2.1 Classe `Ville`

Question 4. Créez une classe `Ville` ; le constructeur doit permettre d'initialiser une ville à l'aide de son nom, et de ses coordonnées `x` et `y`.

Question 5. Ajoutez une méthode `distance_vers(autre_ville)` qui renvoie la distance entre la ville courante et une autre ville passée en paramètre.

Question 6. Implémentez la méthode `__str__` pour que la fonction `print` appelée sur un objet `Ville` affiche le nom de la ville.

2.2 Méthode utilitaires

Question 7. Créez une fonction globale nommée `generer_villes`. Cette fonction prend un paramètre optionnel `nb_villes` (dont la valeur par défaut est `20`) et renvoie une liste d'objets `Ville` aux coordonnées aléatoires (comprises entre 0 et 300) ; le nom de la ville est simplement un numéro.

Question 8. A l'aide du module python `csv`, écrivez une fonction globale nommée `lire_csv`, qui prend un paramètre le nom d'un fichier CSV (sur chaque ligne, on trouve simplement le nom de la ville et ses coordonnées, le tout étant séparé par des virgules). La fonction lit le fichier et renvoie la liste d'objets `Ville` correspondante.

2.3 Classe `Trajet`

Une `Ville` correspond à un gène, et un trajet, c'est-à-dire une suite de n villes, correspond donc à un chromosome.

Question 9. Créez la classe `Trajet` : ses attributs sont une liste de `Ville` et la longueur du trajet correspondant. Le constructeur prend en paramètre **optionnel** une liste de villes :

- en l'absence de liste fournie, on initialise simplement une liste vide
- si une liste est fournie, on génère un trajet aléatoire (regardez le module `random` de Python) à partir de cette liste (attention à ne pas modifier la liste passée en paramètre !)

Question 10. Créez une méthode `calc_longueur` servant à mettre à jour l'attribut `longueur` à chaque modification du trajet (pensez à reutiliser la méthode `distance_vers` de la classe Ville !).

Question 11. Créez une méthode `est_valide` permettant de vérifier qu'une liste de villes correspond bien à un trajet valide (c'est-à-dire que chaque ville n'est présente qu'une et une seule fois).

Question 12. Ecrivez la méthode `__str__` pour que la représentation textuelle d'un trajet soit simplement la liste de ses villes.

2.4 Classe `Population`

A présent que vous pouvez créez des *individus* / *trajets*, vous allez écrire le code permettant de gérer une *population*.

Question 13. Créez la classe `Population` ; une population est simplement une liste de `Trajet` (initialement vide) (ceci nous permet de créer des populations vides auxquelles nous rajouterons par la suite des individus).

Question 14. Ajoutez la méthode `initialiser(taille, liste_villes)` : cette méthode initialise une population de *taille* individus, générés aléatoirement à partir de la liste des villes fournies (il suffit de faire appel au code de `Trajet` déjà écrit !).

Question 15. Ajoutez la méthode `ajouter(trajet)` qui, comme son nom l'indique, ajoute le trajet fourni à la population courante.

Question 16. Ajoutez la méthode `meilleur()` qui retourne le meilleur individu, c'est-à-dire le trajet de plus petite longueur dans la population (indice : comment trier une liste en Python ?).

Question 17. Ecrivez la méthode `__str__` pour que la représentation textuelle d'une population soit simplement la liste des trajets qu'elle contient.

Implémentation de l'algorithme génétique - Classe `PVC_Genetique`

Une classe `PVC_Genetique` est fournie (elle ne contient pour l'instant qu'une méthode utilitaire `clear_term`). Il s'agit de la classe principale de votre programme : c'est par elle qu'on va initialiser et exécuter l'algorithme génétique.

Question 18. Créez le constructeur de cette classe : il prend un paramètre *obligatoire* (la liste des villes) et deux paramètres *optionnels* (la taille de la population, par défaut à `40`, et le nombre de générations, par défaut à `100`).

Question 19. Ecrivez la méthode `croiser(parent1, parent2)`. Elle doit renvoyer l'enfant issu du croisement.

⚠ Il faut bien veiller à ce que l'enfant généré corresponde à un trajet valide (on ne peut pas juste croiser les listes comme dans l'exemple du cours, car on pourrait avoir la même ville présente plusieurs fois dans le trajet) ! Proposez une méthode permettant de vous en assurer.

Question 20. Ecrivez la méthode `muter(trajet)`. Elle doit renvoyer le trajet muté.

⚠ Ici aussi, on ne peut pas simplement remplacer une ville par une autre ville aléatoire. Quelle méthode de mutation proposez-vous pour garantir des trajets valides ?.

⚠ Pensez bien à recalculer la longueur du trajet ainsi modifié !.

Question 21. Ecrivez la méthode `selectionner(population)`, correspondant à une méthode de sélection de votre choix pour les individus qui devront se reproduire (selon ce que vous mettrez en place, elle peut renvoyer directement une liste d'individus, ou bien un seul individus, auquel cas il faudra rappeler plusieurs fois la méthode `selectionner`).

Vous disposez à présent de toutes les briques pour faire évoluer une population !

Question 22. Ecrivez la méthode `évoluer(population)` : cette méthode utilise les méthodes précédentes de la classe `PVC_Genetique` pour faire évoluer une population d'individus, et retourne la nouvelle population.

Question 23. Ajoutez au constructeur deux paramètres optionnels : `elitisme = True` et `mut_proba = 0.3`, et modifiez votre compte pour prendre en compte l'élitisme lors de la sélection, et la valeur du paramètre `mut_proba` avant d'effectuer une mutation.

Question 24. Enfin, la méthode principale de la classe `PVC_Genetique` est la méthode `executer`. C'est elle qui crée la population initiale puis la fait évoluer sur le nombre de générations spécifié lors de l'appel au constructeur. Elle se charge également de conserver la trace du meilleur trajet trouvé.

💡 Vous pouvez, pour chaque génération, afficher le meilleur individu de cette génération ainsi que le meilleur individu depuis le début. Pour ne pas "polluer" votre terminal, vous pouvez utiliser la fonction `clear_term` fournie.

Une classe `PVC_Genetique_GUI` vous est fournie dans le fichier `GeneticTSPGui.py` ; cette classe contient tout le code nécessaire pour une représentation "graphique" de vos trajets.

Question 25. Pour l'utiliser, vous devez :

- ajouter un attribut de type `PVC_Genetique_GUI` dans la classe `PVC_Genetique` ; le constructeur a besoin de connaître la liste des villes pour les afficher ;
- ajouter dans votre méthode `executer` un attribut optionnel `afficher = True`

- quand l'attribut **`afficher`** vaut **`True`**, faire appel à la méthode **`affiche()`** du **`PVC_Genetique_GUI`** ; cette méthode prend en paramètre le meilleur trajet global, ainsi que le meilleur trajet de la génération courante, et un paramètre booléen optionnel **`afficher_noms`** qui permet d'afficher les noms des villes sur les cartes.
- ⚠ A la fin de la méthode **`exécuter`**, ajouter la ligne ***`votre_objet_GUI`*`.window.mainloop()`**.