Projet RO 2022, un dernier tour en Côte d'Or

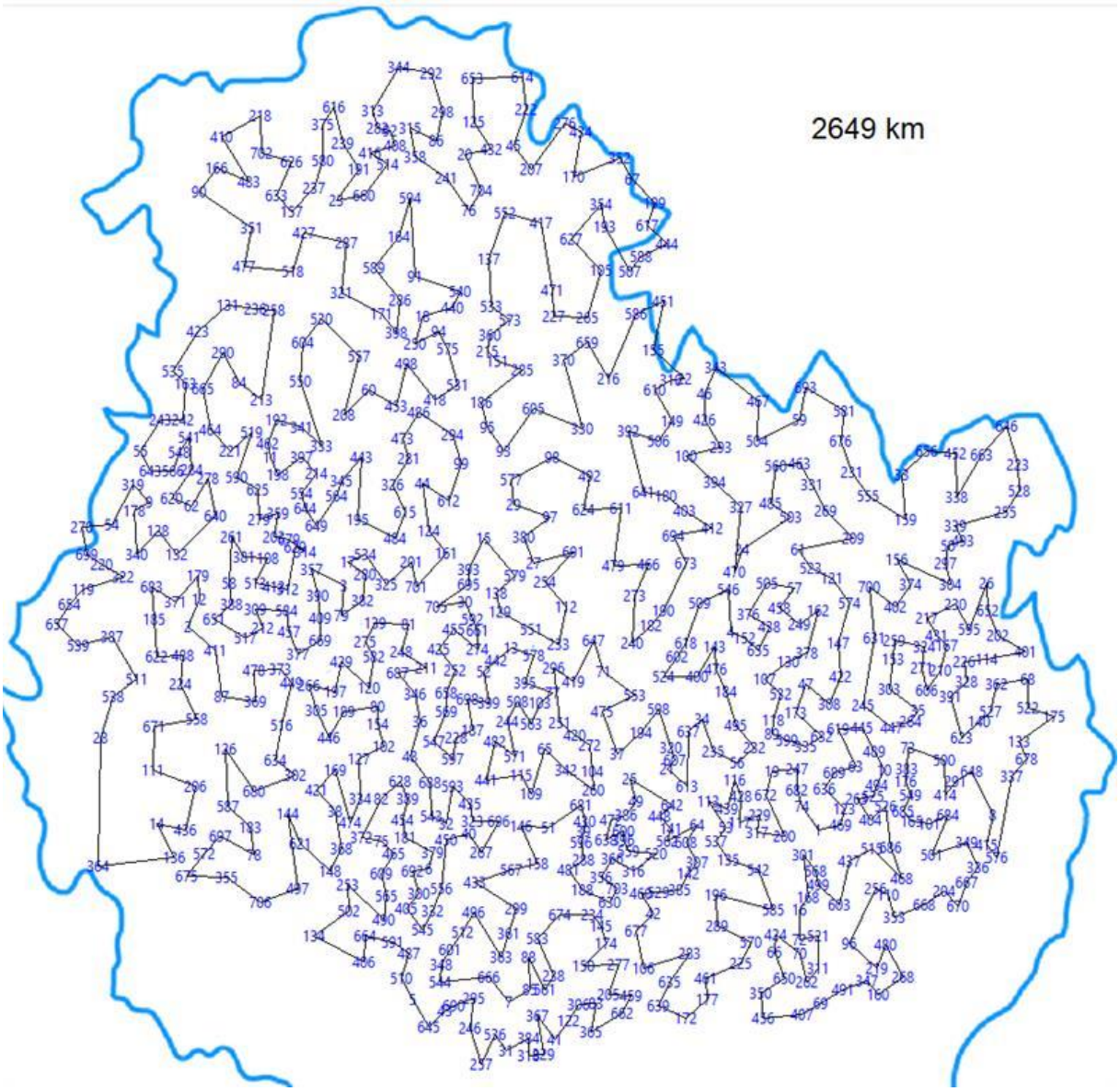


Table des matières

I. Description générale

A. Définition du problème

À la suite de la pandémie de COVID-19 et aux difficultés qui sont apparues dans la distribution des masques et des vaccins, nous souhaitons anticiper la prochaine pandémie et envisager un meilleur mode de distribution du matériel médical. Nous lançons une étude de faisabilité de livraison de matériels par drone ou hélicoptère, par exemple pour un parachutage des doses de vaccins au-dessus de chaque commune du département. Pour cela, nous devons déterminer l'autonomie nécessaire pour chaque appareil, et donc le nombre de kilomètres total à parcourir pour desservir l'ensemble des communes.

Plusieurs possibilités sont envisagées :

- Livraison en plusieurs tournées dont une première tournée pour les 70 villes les plus peuplées
- Livraison en plusieurs tournées dont une tournée pour l'ensemble des villages et hameaux de moins de 100 habitants.
- Livraison par un seul appareil des 706 villes du département

B. Les Données

Nous avons à notre disposition 3 fichiers de données au format texte avec 1 ligne par commune, sous la forme suivante :

NumVille NomVille Latitude Longitude

Instances à traiter :

- *Top80.txt* pour les 80 communes les plus peuplées,
- *moinsDe100.txt* pour les communes de moins de 100 habitants,
- *ListeComplete.txt* pour l'ensemble des communes de Côte d'Or.

C. Résultats attendus

Pour chaque fichier de données, on veut obtenir une tournée optimale, c'est-à-dire une liste ordonnée de tous les numéros de ville, de manière à minimiser la distance totale parcourue.

Le langage de programmation est libre, aucune interface graphique n'est demandée.

D. Méthodologie

Nous allons procéder par étapes en améliorant la qualité de nos solutions au fur et à mesure. Nous prendrons systématiquement comme jeu de données exemple le fichier top80.txt.

II. Obtention d'une première solution (TP1)

1. Lire le fichier *top80.txt* qui contient le jeu de données et créer votre **ListeVilles**. Chaque **Ville** doit contenir au moins NumVille Latitude et Longitude.
2. Ecrire une procédure qui affiche les Villes de votre ListeVilles et vérifier que cela correspond bien aux données du fichier de départ.
3. Ecrire une fonction de distance **d(ville v1,ville v2): float ;** qui calcule la distance entre 2 villes.

Formule Haversine utilisée pour le calcul des distances :

*D(v1,v2) = abs(r*acos((sin(y1)*sin(y2)) + (cos(y1)*cos(y2)*cos(x1-x2))))*

où x1,y1 et x2,y2 sont les coordonnées des villes v1 et v2 exprimées en radians , et r=6371 (x = longitude, y = latitude)

---

4. Vérifier que vous obtenez la bonne distance entre la ville 1 et la ville 2 en l'affichant:

```
distance entre ville 1 et ville 2 :  34.425096765231004  km
```

5. Une **tournée** est une liste de villes. Créer une première tournée croissante à partir de la ListeVilles. (les villes sont dans l'ordre croissant de leur numéro)
6. Ecrire une procédure **afficheTour(tour T) ;** pour afficher les numéros des villes de la tournée et vérifier que le résultat est similaire à :

```
tournée croissante [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80]
```

7. Ecrire une fonction **cout(T) : float ;** qui calcule la distance totale de la tournée. N'oubliez pas de rentrer à la maison à la fin !
8. Demander à l'enseignant pour vérification de la distance obtenue
9. Ecrire la fonction **tourAleatoire(ListeVille L) :tour ;** qui produit une tournée aléatoire.

III. Méthodes gloutonnes (TP2)

Plusieurs méthodes gloutonnes peuvent être testées. L'objectif ici est d'obtenir rapidement une tournée de qualité moyenne, que l'on utilisera comme *Solution Initiale* dans les étapes suivantes.

A. Méthode gloutonne « plus proche voisin »

Principe de l'algorithme : on construit la tournée en piochant dans la liste des villes « non visitées » la plus proche de la dernière ville ajoutée à la tournée. On commence en ajoutant la ville numéro 1 à une tournée vide. Toutes les autres villes sont « non visitées ». A la fin de l'algorithme toutes les villes sont « visitées » et font partie de la tournée résultat.

```
Fonction plus_proche_voisin(ville s) : tour T1
Pour toute ville v,
      Visité[ville] = faux
T1 = [s]
Visité[s] = vrai
Tant Que il existe un sommet non visité
      Suivant=plus_proche(s)  //cette fonction doit retourner la ville non
                                visitée la plus proche de s
      visité[suivant]=vrai
      T1.append(suivant)
      s = suivant
Fin Tant que
Retourner T1
```

Variante : Glouton plus_proche_voisin amélioré

Le résultat de la méthode proche_voisin dépend directement du sommet de départ choisi. On peut construire une fonction qui lance l'exécution de plus_proche_voisin() à partir de chacune des villes de départ possibles, et renvoie la meilleure solution trouvée.

B. Méthode gloutonne « insertion proche »

Principe de l'algorithme : Puisque toutes les villes doivent appartenir à la tournée résultat, les deux plus éloignées aussi. On commence donc la tournée en intégrant les 2 villes qui sont les plus éloignées. Puis on intègre successivement à la tournée la ville non visitée qui est la plus proche de la tournée existante. On ajoute cette ville au bon endroit dans la tournée (pas forcément à la fin).

```
Fonction insertion_proche() :tournée T2
Pour toute ville v,
      Visité[v] = faux
Choisir v1 et v2 les deux villes les plus éloignées l'une de l'autre
T2  = [v1, v2]
Visité[v1]=vrai
Visité[v2]=vrai
Tant Que  il existe un sommet v non visité
            Suivant = sommet le plus proche de T2 // on calcule la distance ajoutée à la tournée par
      l'insertion de v entre i et i+1

            T2.append(Suivant)
      Visité[suivant] <- vrai
Fin Tant Que
Retourner T2
```

C. Méthode gloutonne « insertion loin »

Principe de l'algorithme : Puisque toutes les villes doivent appartenir à la tournée résultat, les deux plus éloignées aussi. On commence donc la tournée en intégrant les 2 villes qui sont les plus éloignées. On poursuit avec le même principe pour la suite : on intègre successivement à la tournée la ville non visitée dont la plus courte distance par rapport à la tournée existante est la plus grande.

```
Fonction insertion_loin() :tournée T3
Pour toute ville v,
      Visité[v] = faux
Choisir v1 et v2 les deux villes les plus éloignées l'une de l'autre
T3  = [v1, v2]
Visité[v1]=vrai
Visité[v2]=vrai
Tant Que  il existe un sommet v non visité
      Suivant = sommet le plus loin de T3 // pour tout v on calcule Dmin distance minimale ajoutée à la tournée par l'insertion de v, on choisit ensuite le sommet avec le plus grand Dmin
      T3.append(Suivant)
      Visité[suivant] <- vrai
Fin Tant Que
Retourner T3
```

IV. Recherche locale (TP3)

Vous allez améliorer votre solution initiale de façon itérative, en cherchant dans le voisinage de la solution courante si une meilleure solution existe. Cette solution devient la nouvelle solution courante, dont on examine le voisinage. On répète le processus jusqu'à avoir une solution qui est meilleure que toutes ses voisines.

L'algorithme de base est le même pour chacune des recherches locales. La seule partie qui diffère est l'exploration du voisinage. Dans chaque fonction, on implémente un voisinage différent. Les principales difficultés portent sur la gestion du début et de la fin de la tournée, ainsi que sur la copie de la tournée voisine comme nouvelle tournée courante.

```
Fonction recherche_locale(tournée T_entrée):tournée T_sortie)
Tcourante = T_entrée
Fini=Faux
Tant que (Fini==Faux)
	Fini=Vrai
	Explorer le voisinage de Tcourante :  //dépend du voisinage choisi
	Si cout_tournée(Tvoisin) < cout_tournée(Tcourante)Alors
		Tcourante=Tvoisin  //nécessaire de faire une vraie copie
		Fini=Faux
	FinSi
Fin Tant Que
Retourner Tcourante
```

A. Echange de successeurs

1. Principe « premier d'abord »

Le voisinage est l'ensemble des tournées obtenues par inversion de deux villes consécutives dans la tournée. Dès qu'on trouve une amélioration, le voisin devient la tournée courante.

```
Exploration_successeurs_premier_d_abord
Pour toute position i dans Tcourante
	Tvoisin = Echanger(Tcourante,i,i+1)
```

Autre façon de faire, en remplaçant la création de Tvoisin par le calcul direct du gain potentiel :

```
Pour toute position i dans Tcourante
Si d(i-1,i)+d(i+1,i+2) > d(i-1,i+1)+d(i,i+2) Alors
Tcourante = Echanger(Tcourante,i,i+1)
```

2. Principe « meilleur d'abord »

Le voisinage est l'ensemble des tournées obtenues par inversion de deux villes consécutives dans la tournée. On attend d'avoir parcouru tout le voisinage pour choisir le meilleur voisin comme nouvelle tournée courante.

```
Exploration_successeurs_meilleur_d_abord
Pour toute position i dans Tcourante
	Tvoisin = Echanger(Tcourante,i,i+1)
	Mémoriser le Meilleur_voisin et le meilleur_cout associé
Tcourante = Meilleur_voisin
```

B. Echange de sommets quelconques

1. Principe « premier d'abord »

Le voisinage est l'ensemble des tournées obtenues par inversion de deux villes i et j quelconques dans la tournée. Dès qu'on trouve une amélioration, le voisin devient la tournée courante.

```
Exploration_ij_premier_d_abord
Pour toute position i dans Tcourante
      Pour toute position j dans Tcourante
            Tvoisin = Echanger(Tcourante,i,j)
```

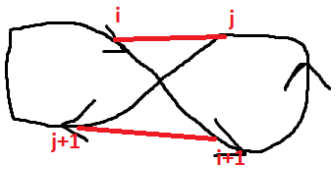
2. Principe « meilleur d'abord »

Le voisinage est l'ensemble des tournées obtenues par inversion de deux villes i et j quelconques dans la tournée. On attend d'avoir parcouru tout le voisinage pour choisir le meilleur voisin comme nouvelle tournée courante.

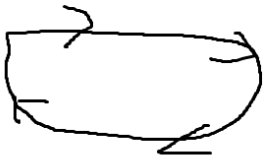
```
Exploration_ij_meilleur_d_abord
Pour toute position i dans Tcourante
      Pour toute position j dans Tcourante
            Tvoisin = Echanger(Tcourante,i,j)
            Mémoriser le Meilleur_voisin et le meilleur_cout associé
Tcourante = Meilleur_voisin
```

C. Echange 2-opt

Le voisinage est l'ensemble des tournées obtenues par inversion d'un tronçon de tournée, entre les villes i et j+1, comme indiqué sur le schéma ci-contre. On cherche ainsi à « démêler » les éventuels nœuds d'une tournée, en parcourant toute une partie de la tournée en sens inverse.



Là encore 2 possibilités:



- créer la tournée Tvoisine, calculer son cout, et remplacer Tcourante si le cout est meilleur
- calculer le gain potentiel du 2opt entre i et j et modifier Tcourante seulement si le gain est positif. Gain=d(i, i+1)+ d(j, j+1) -d(i,j)-d(i+1, j+1)puisque la partie de la tournée inversée garde la même longueur.

1. Principe « premier d'abord »

```
Exploration_2opt_premier_d_abord
Pour toute position i dans Tcourante
   Pour toute position j > i+1 dans Tcourante
      Si d(i, i+1)+ d(j, j+1) > d(i,j)+d(i+1, j+1)
            Tcourante = Retourner(Tcourante,i+1,j)
// on doit retourner toute la partie de tournée entre i+1 et j, pas seulement inverser les extrémités !
```

2. Principe « meilleur d'abord »

```
Exploration_2opt_meilleur_d_abord
Pour toute position i dans Tcourante
   Pour toute position j > i+1 dans Tcourante
      Si d(i, i+1)+ d(j, j+1) > d(i,j)+d(i+1, j+1)
            Mémoriser le Best_cout, et les indices Best_i Best_j
Tcourante = Retourner(Tcourante,Best_i+1,Best_j)
```

D. Variantes et Combinaison de Recherches locales

Les recherches locales implémentées peuvent être combinées tant que l'une d'elles apporte une amélioration à la solution courante.

Exemple d'algorithme :

```
Ma_super_recherche_locale(tournée T)
Continer=True
Meilleur=T
Tant que Continuer
       Continuer=False
       T=recherche_locale_ij_meilleur_d_abord(T)
       T=recherche_locale_2opt_premier_d_abord(T)
       Si cout(Meilleur)<cout(T)
              Meilleur=T
              Continuer=True
       FinSi
FinTant Que
```

En choisissant une autre tournée que glouton_proche_voisin(ville1), on peut obtenir de meilleurs résultats. Parfois c'est avec une moins bonne solution initiale que la tournée obtenue après recherche locale est meilleure. Les possibilités de combinaisons entre algorithmes gloutons et recherches_locales sont multiples… Faites des essais !

V. Algorithme Génétique (TP4)

A. Principe général

Au lieu de manipuler une seule tournée à la fois, on fait évoluer un ensemble de tournées, sur lesquelles on applique des croisements et des mutations

Trame de l'algorithme

```
Constituer X solutions de départ avec tournée_aléatoire()
génération = 1
Tant que génération < NBMAX :
      Choisir N parents parmi les X solutions
      Croiser les parents 2 par 2 pour obtenir N enfants
      Muter y<<N enfants en leur appliquant une recherche_locale
      Choisir N solutions parmi les N parents + N enfants
      Génération++
Fin Tant que
Renvoyer la meilleure solution trouvée
```

Tous les détails de l'algorithme ont été présentés dans le CM8.

B. Indications complémentaires

1. Comment croiser les tournées P1 et P2 ?

E1 := P1[0 :Pt_de_croisement]

E2 := P2[0 :Pt_de_croisement]

Ajouter ensuite dans E1 les villes de P2[Pt_de_croisement :N-1] qui ne sont pas déjà dans E1, puis compléter avec les villes de P1[Pt_de_croisement :N-1] qui ne sont pas déjà dans E1.

Exemple de croisement de P1=[1,2,3,4,5] et P2=[3,1,5,4,2]

Si pt_croisement=2 P1=[1,2,3,4,5] et P2=[3,1,5,4,2]

On commence à créer E1 avec le début de P1 :

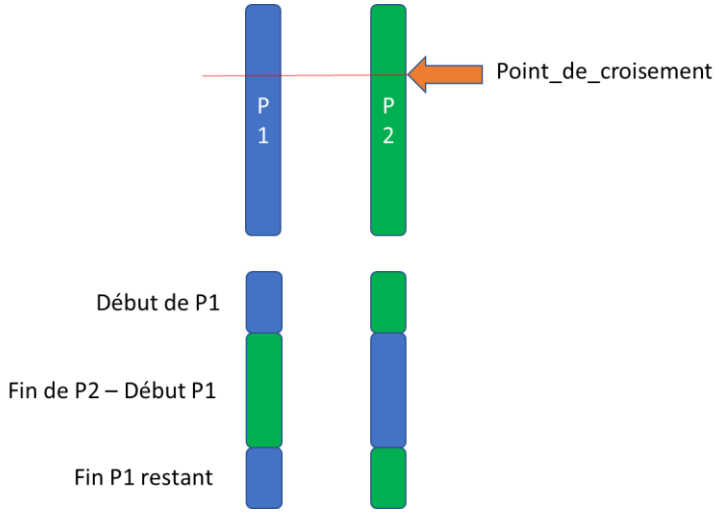
E1=[1,2] E2=[3,1]

Puis on ajoute à E1 la fin de P2 qui n'est pas dans E1

E1=[1,2,5,4] E2=[3,1,4,5]

Enfin on complète avec la fin de P1 (ou le début de P2)

E1=[1,2,5,4,3] E2=[3,1,4,5,2]



2. Exemple de paramétrage de l'AG

De nombreux paramètres sont à choisir vous-même. Si vous êtes perdu, vous pouvez tester avec ceux-là :

Taille_population X=800

Solutions de départ aléatoires : tour_aleatoire()

Nb_max_générations NBMAX=50

Nb_parents N = 400

Je choisis les 200 meilleurs + 200 au hasard

Je croise les parents 2 par 2 au hasard

Point de croisement = 10

On obtient Nb_enfants=Nb_parents=400

Nb_mutations y=40 .

J'applique la recherche_locale_2opt sur les 20 meilleurs enfants +20 au hasard

Je lance aussi 1 super_mutation (combinaison de recherches locales) sur la meilleure solution

J'ajoute les enfants à ma population et je sélectionne les 20 meilleurs individus + les autres au hasard