Le problème du voyageur de commerce

Paul Dufour
Numéro d’inscription : 48121

Plan

I - Introduction

II- Méthodes de résolution classiques [partie commune]

III–Les algorithmes génétiques [Initiative personnelle]

1- Description

2- Optimisation

3- Comparaison

I - Introduction

- **Problème du voyageur de commerce:** « Quel est le plus court chemin qui passe par toutes les villes une fois et qui revient à la ville de départ ? »
- **But:** recherche de solutions au problème du voyageur de commerce et modélisation de celles-ci
- **Applications multiples**: fabrication de processeurs, bus scolaires …
- Problème NP-Complet : algorithmes d'optimisation

Localisation des écoles, des centres sportifs et des dépôts de bus

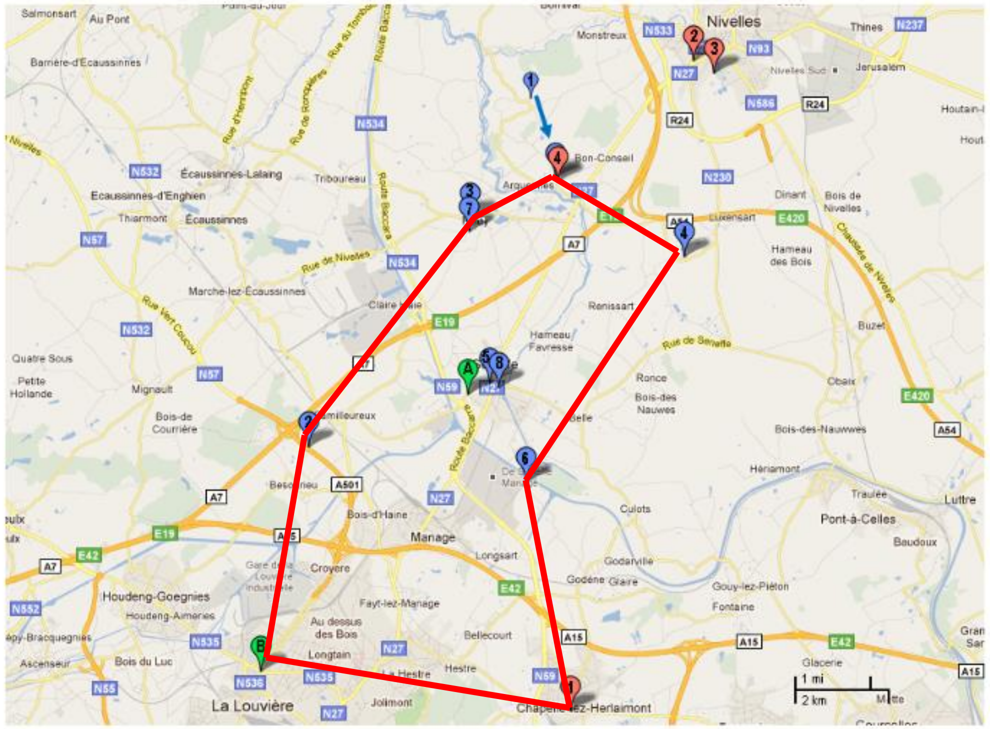


Figure 1: Exemple pour la commune de Seneffe (Belgique)

I - Méthodes de résolution classiques [partie commune]

Modélisation du problème en python

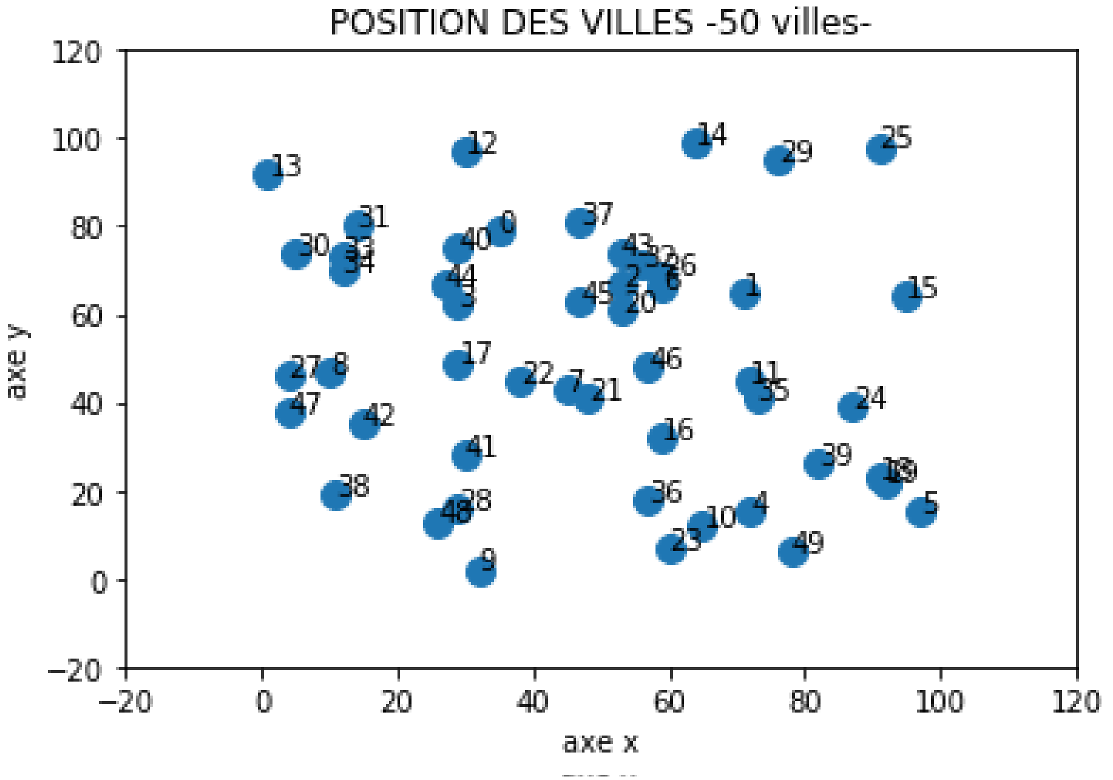


Figure 1: Graphique représentant la position des villes

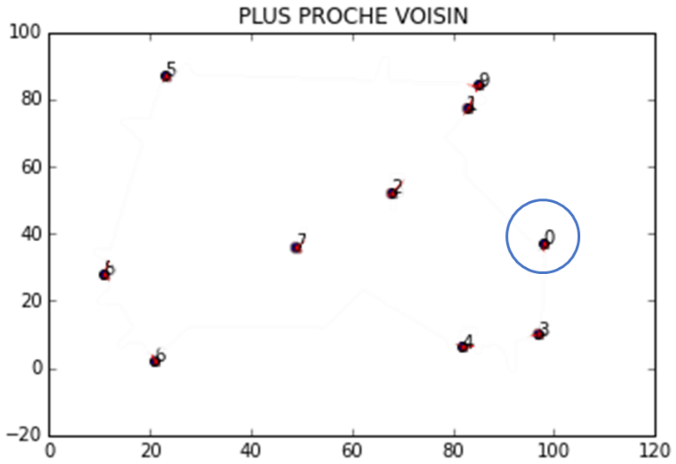
- Carré de dimension 100x100 km
- 50 villes réparties aléatoirement et uniformément.
- On attribue un numéro à chaque ville.
- Exemple d'un chemin possible: [0,1,4,5,6,9,7,2,4,0]

Méthode naïve: essayer tout les chemins possibles et les comparer.

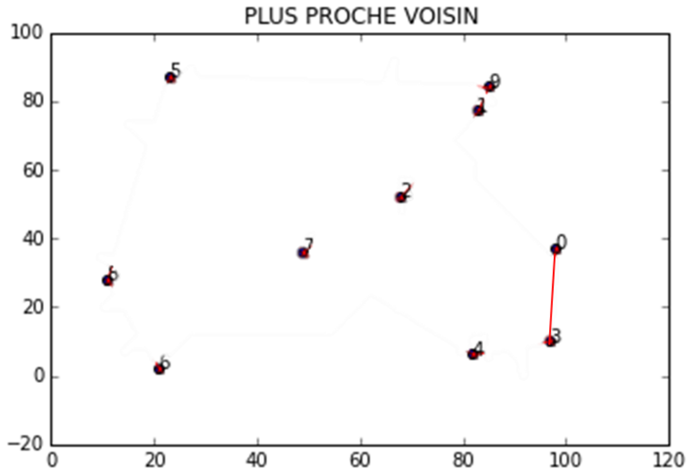
Défaut: Explosion combinatoire: Complexité en O(n!)

Algorithme de type glouton- Méthode du plus proche voisin

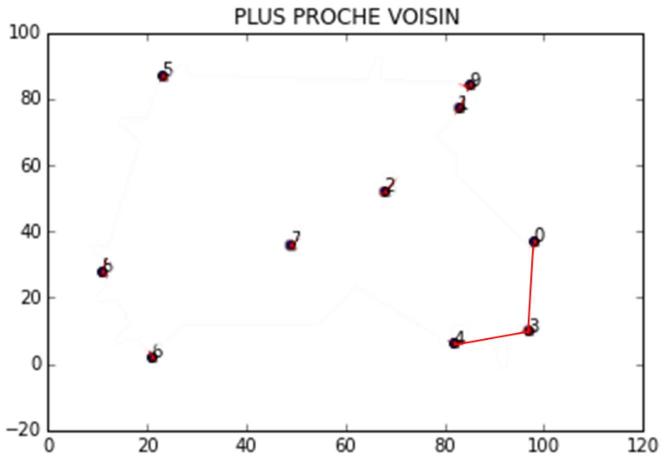
Idée: Créer une solution en recherchant à chaque fois le voisin le plus proche de la ville actuelle.



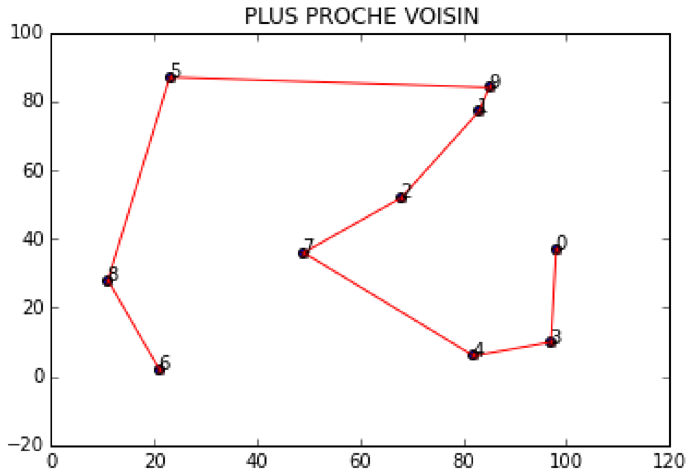
[0]



[0, 3]



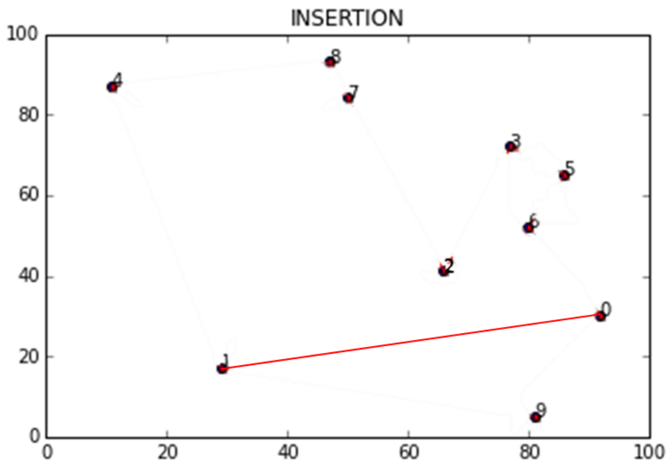
[0, 3, 4]



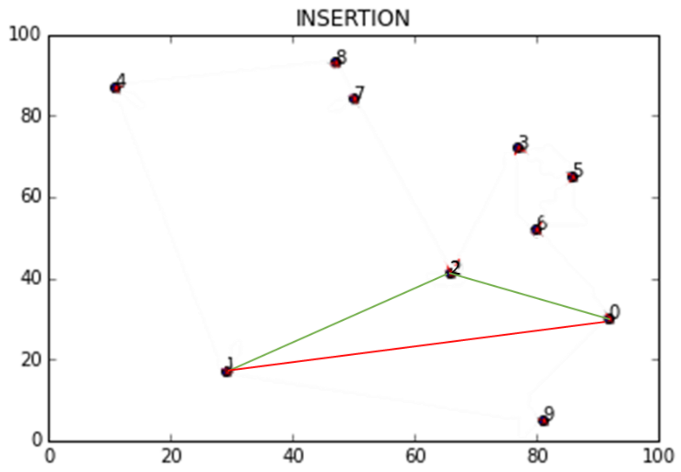
[0, 3, 4, 7, 2, 1, 9, 5, 8, 6, 0]

Algorithme de type glouton- Méthode de l'insertion

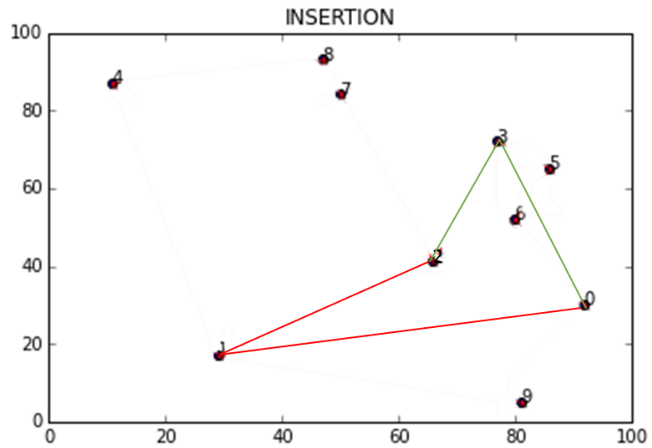
Idée: On insère une nouvelle ville de manière à ce qu'elle augmente au minimum la longueur totale.



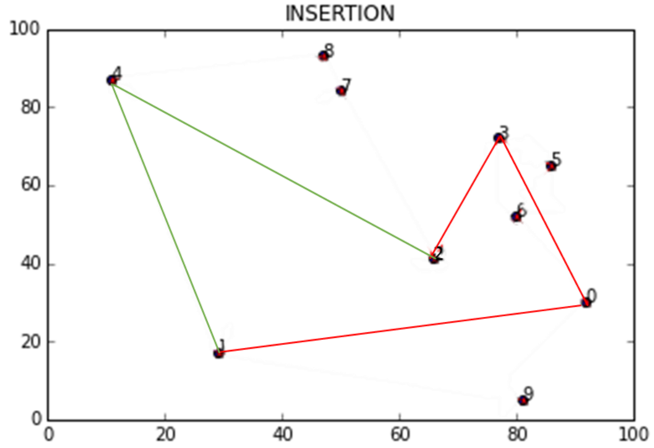
[0, 1, 0]



[0, 2, 1, 0]

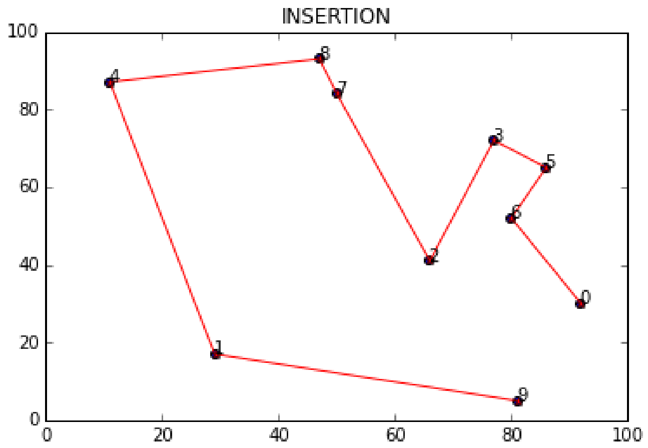


[0, 3, 2, 1, 0]



[0, 3, 2, 1, 4, 0]

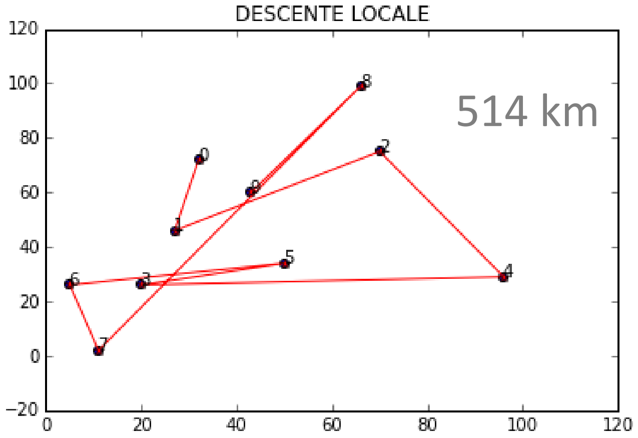
o o o



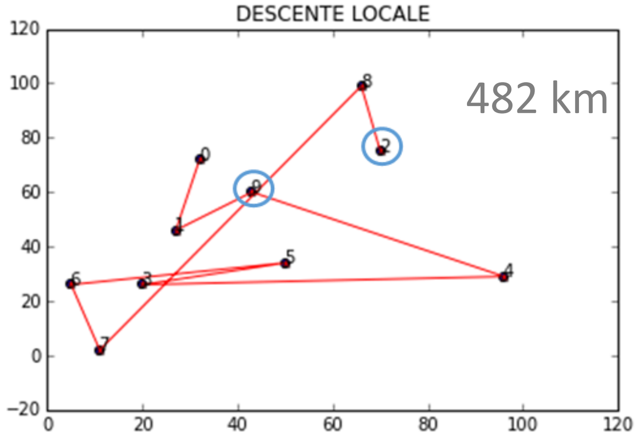
[0, 6, 5, 3, 2, 7, 8, 4, 1, 9, 0]

Amélioration d’une solution - Méthode de la descente locale

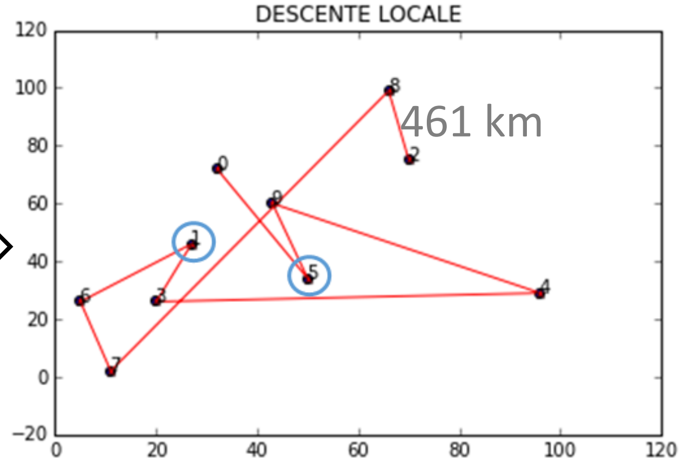
Idée: On permute aléatoirement deux villes d’une solution initiale et on garde le meilleur chemin.



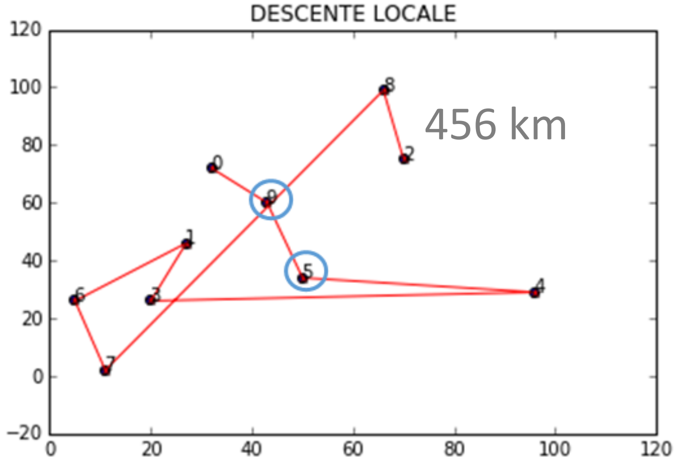
[0, 1, 2, 4, 3, 5, 6, 7, 8, 9, 0]



[0, 1, 9, 4, 3, 5, 6, 7, 8, 2, 0]

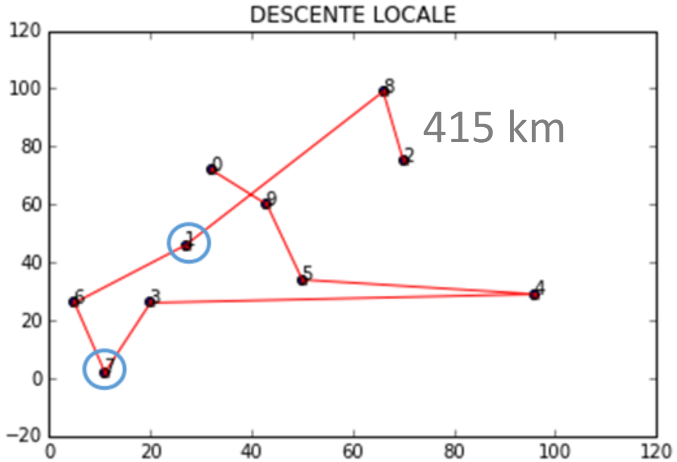


[0, 5, 9, 4, 3, 1, 6, 7, 8, 2, 0]



[0, 9, 5, 4, 3, 1, 6, 7, 8, 2, 0]

2 000*n permutations aléatoires



[0, 9, 5, 4, 3, 7, 6, 1, 8, 2, 0]

II –Les algorithmes génétiques

[Initiative personnelle]

1- Description

- Algorithmes itératifs et stochastiques
- Principes de l'évolution: sélection naturelle et recombinaison génétique

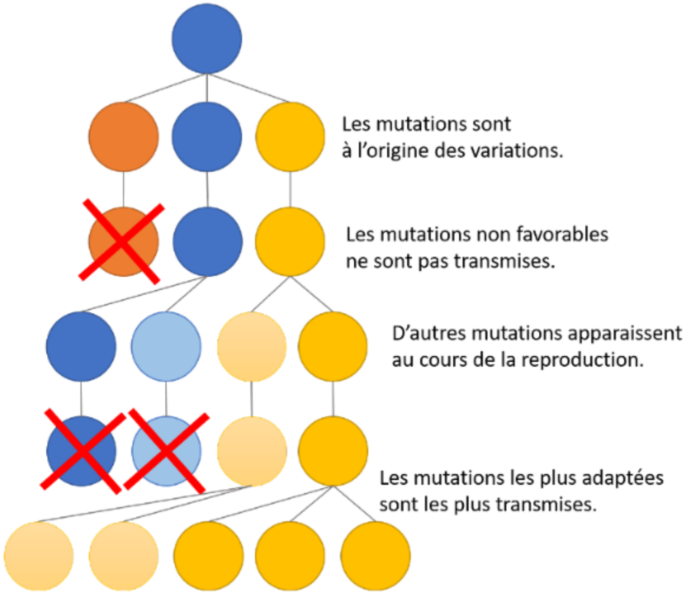
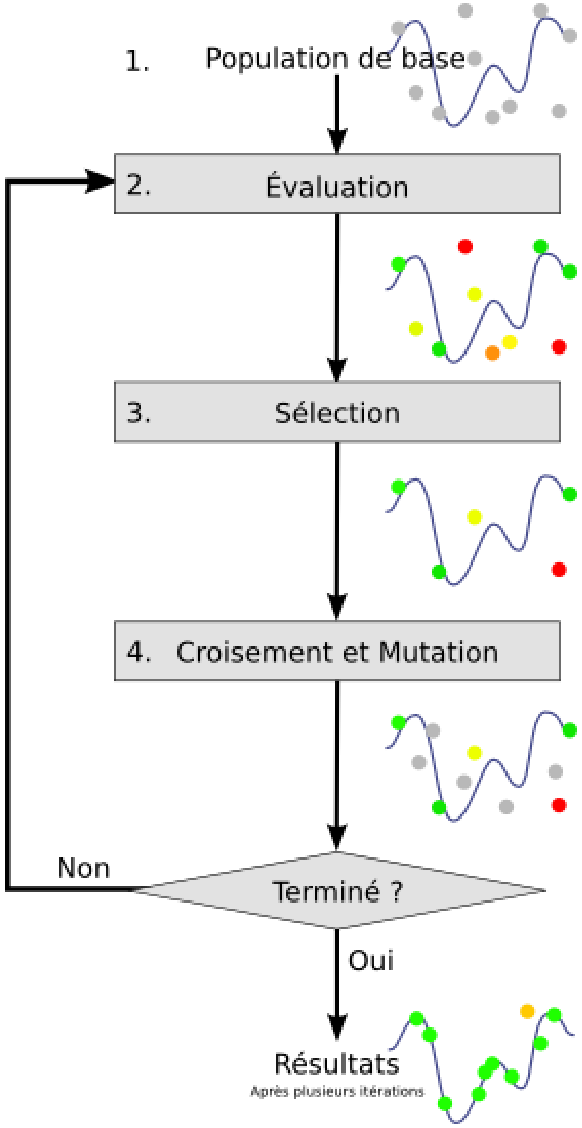


Figure 2 – Le néodarwinisme illustré

Définition	Cas du problème du voyageur de commerce
Individu: solution du problème	Liste de villes par lesquelles passe le voyageur
Population: ensemble d'individus	Liste de listes
Fonction d'adaptation (fitness) : mesure de la qualité de l'individu	Distance totale parcourue par le voyageur
Opérateurs de reproduction: permet de faire évoluer la population	Croisement et mutation
Génération: population à un moment donné du processus	



Génération de chemins aléatoires

Figure 3: Fonctionnement d'un algorithme génétique

Sélection des individus (parents)

Sélection par roulette

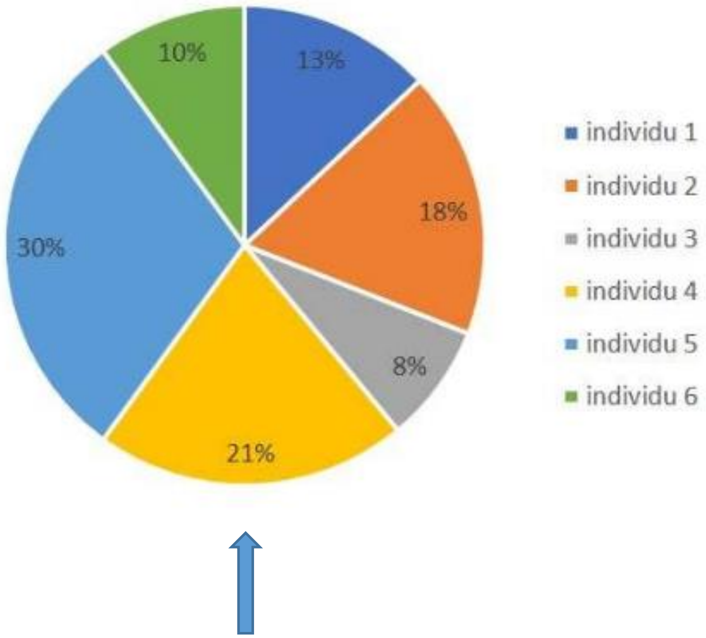


Figure 4 : Sélection par roulette

Sélection par tournoi

[0,1,2,3,0] Combat [0,3,1,2,0]

170 km 120 km

[0,3,1,2,0]

120 km

Probabilité p de sélectionner le vainqueur

[0,1,2,3,0]

170 km

Croisement: Création d’un individu fils

Idée: On génère un fils en croisant les deux parents.

Croisement simple

Croisement

[0,1,2,3,4,0] [0,2,3,4,1,0]

père mère

[0,1,2,4,3,0]

fils

Croisement double

Croisement

[0,1,2,3,4,0] [0,2,3,1,4,0]

père mère

[0,1,3,2,4,0]

fils

Mutations

Mutation avec permutation

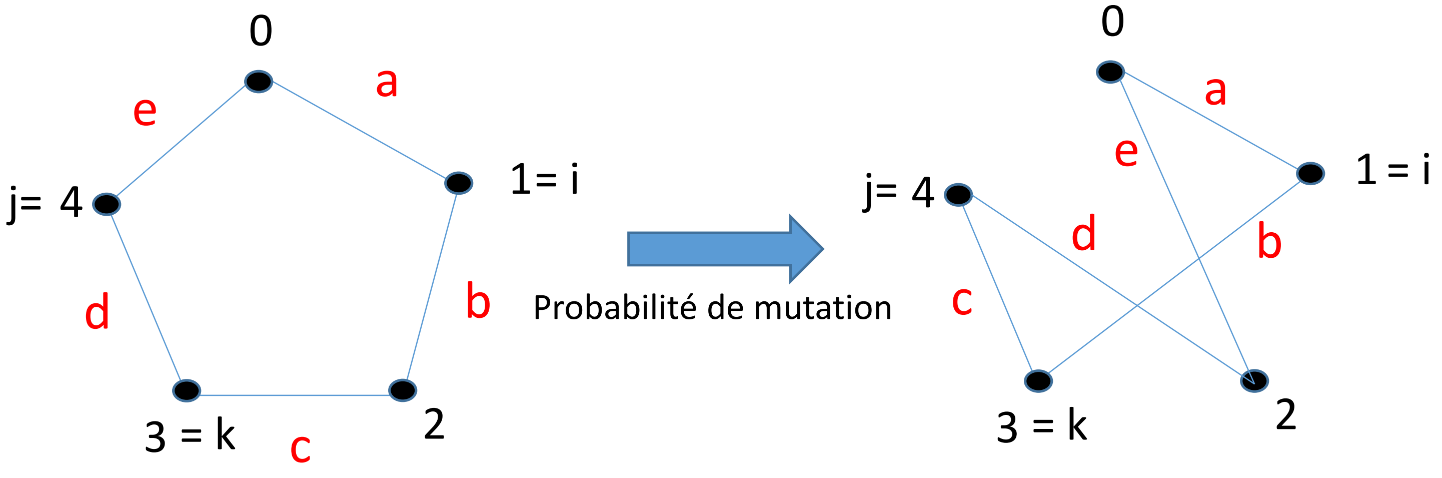
fils
[0,1,2,3,4,0]

Probabilité de mutation

[0,2,1,3,4,0]
fils_muté

Mutation avec double pont séquentiel

Chemin 0 … i,i+1 … k,k+1 … j,j+1 … 0
transformé en 0 … i,k … j,i+1 … k-1,j+1 … 0



[0,1,2,3,4,0] [0,1,3,4,2,0]

Optimisation des individus

Optimisation par inversion du parcours
(appliqué au fils et systématique)

fils_muté

[0,1,2,3,4,5,0]

[0,1,2,3,4,5,0]
fils_muté
170 km

Combat

[0,1,4,3,2,5,0]
fils_optimisé
120 km

[0,1,4,3,2,5,0]
120 km

Insertion de l'individu dans la population

population [[0,1,2,3,4,0], [0,2,4,1,3,0], [0,1,3,2,4,0]]

Idée: Insertion du fils et suppression du pire individu

[0,1,4,2,3,0]

fils

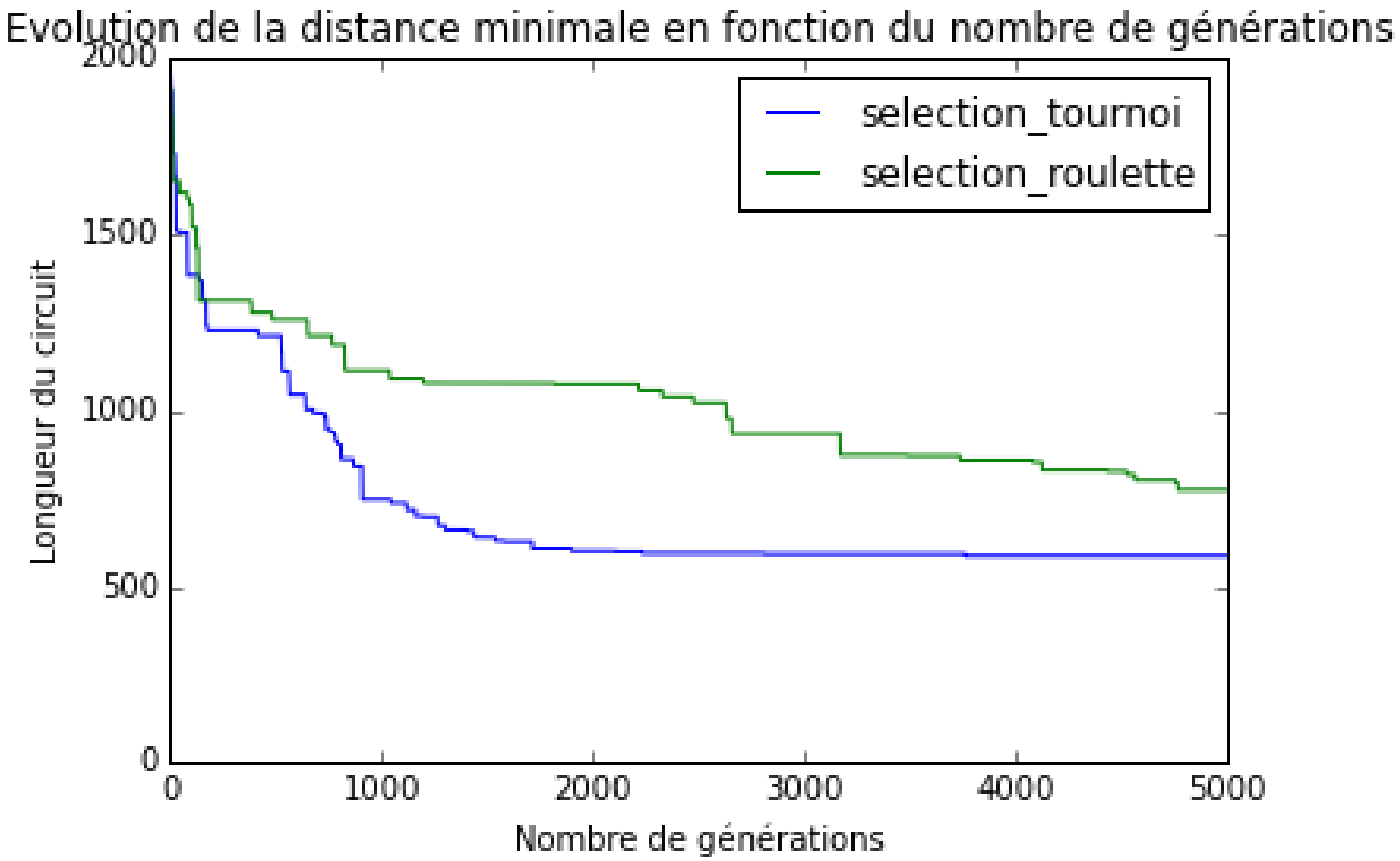
population triée [[0,2,4,1,3,0], [0,1,2,3,4,0], [0,1,4,2,3,0] , ~~[0,1,3,2,4,0]~~]

125 km 150 km 175 km ~~180 km~~

population finale [[0,2,4,1,3,0], [0,1,2,3,4,0], [0,1,4,2,3,0]]

2- Optimisation

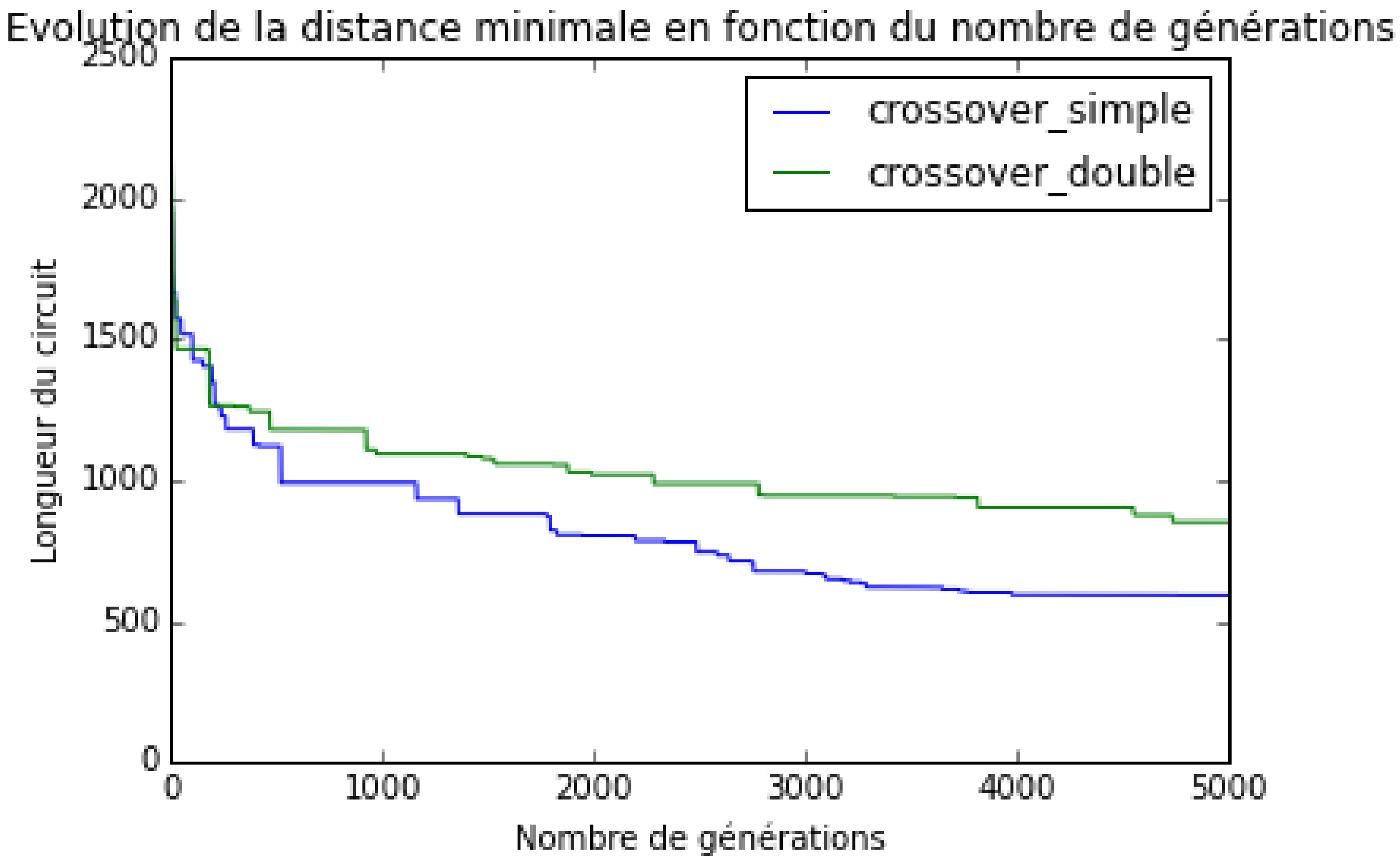
Type de sélection



Résultat: sélection par tournoi plus efficace que celle par roulette

Cause: Forte variabilité génétique pour la sélection par roulette
-> Convergence lente

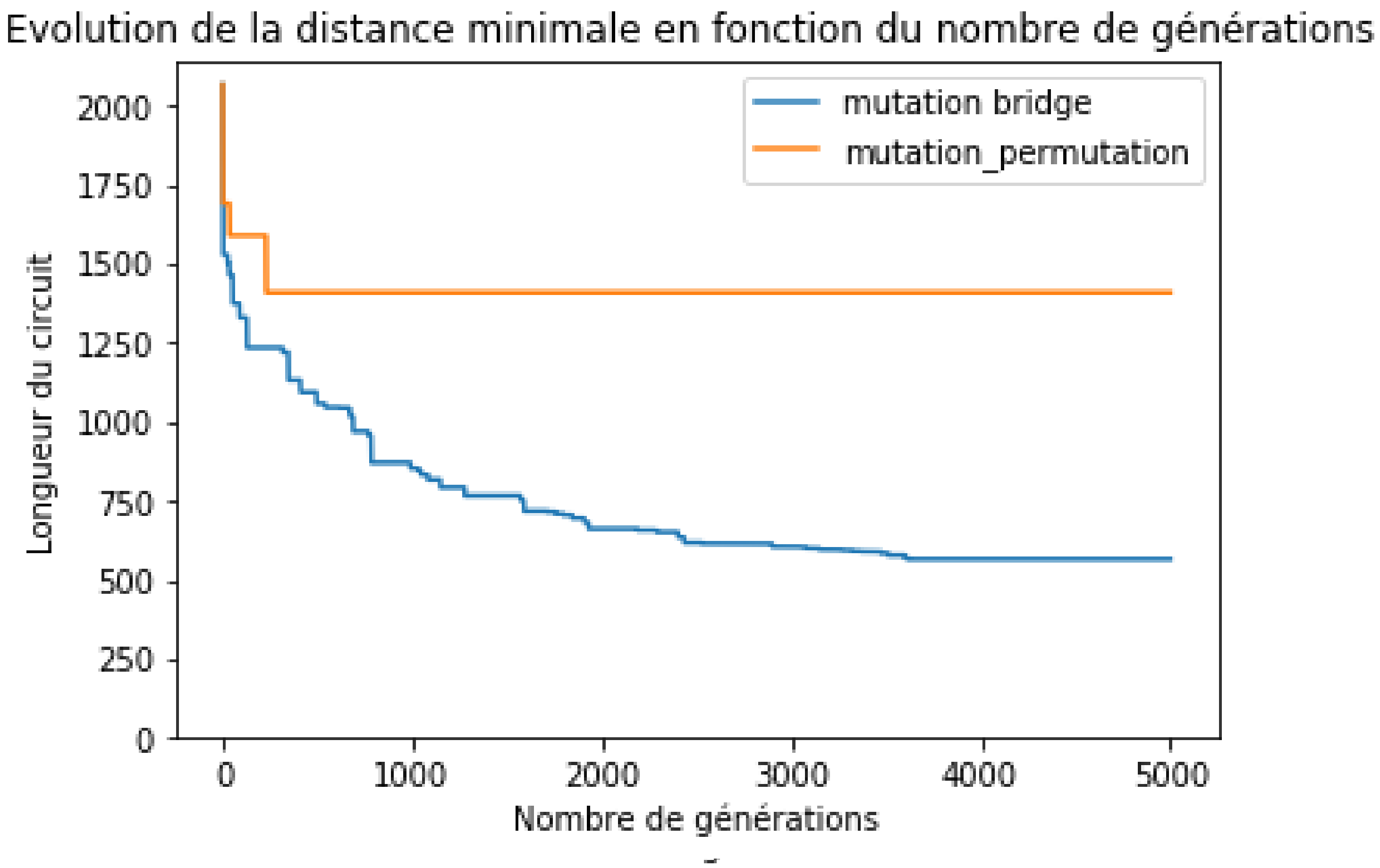
Type de croisement



Résultat: croisement simple plus efficace que croisement double

Cause: Forte variabilité génétique pour le croisement double
-> Convergence lente

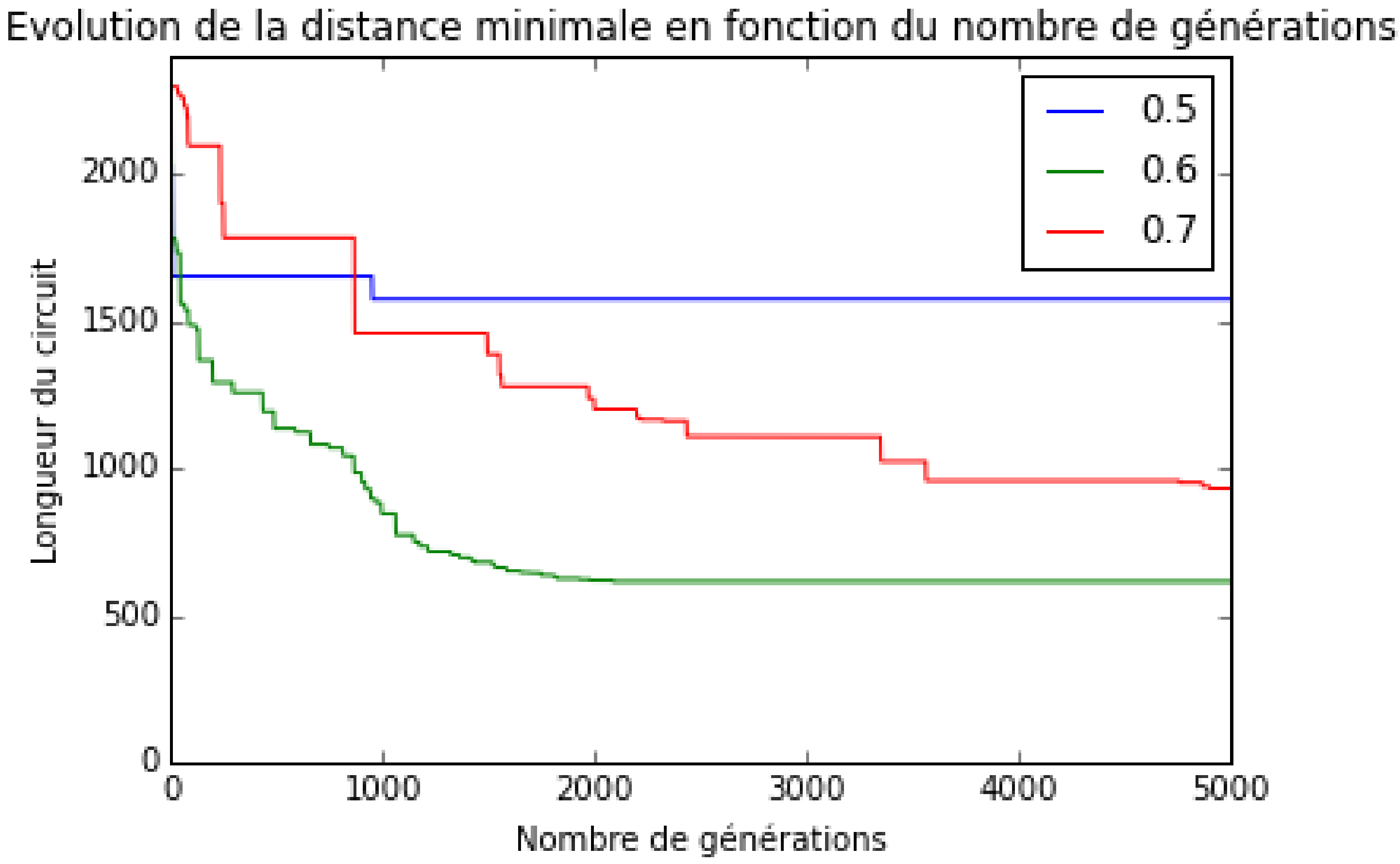
Type de mutation



Résultat: mutation par ponts séquentiels plus efficace que mutation par permutation

Cause: Faible variabilité génétique pour la mutation avec permutation
->Convergence prématurée

Paramètre: taux de mutation



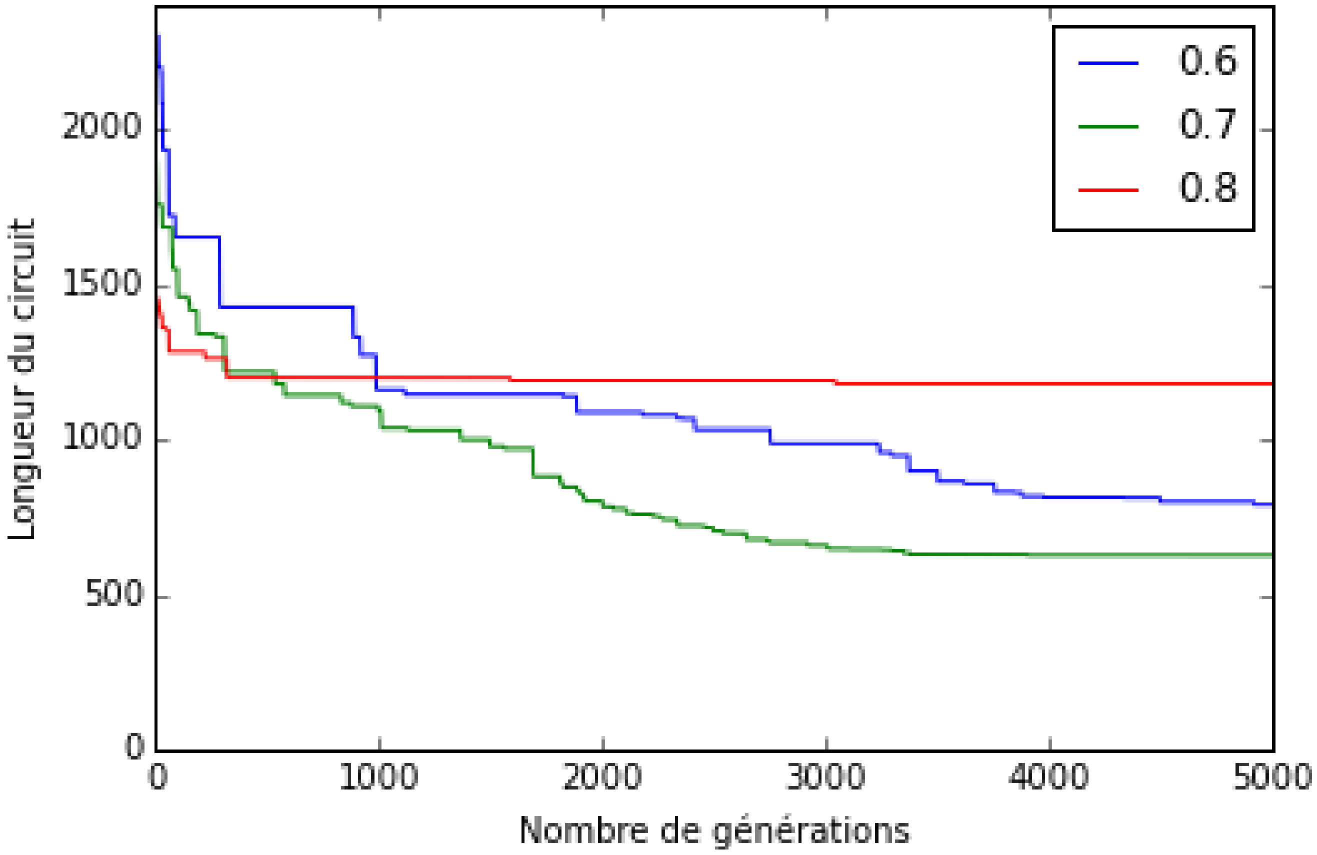
Résultat: taux de mutation optimal: 0.6

Cause:

- taux de mutation = 0.5: peu de mutations donc faible variabilité génétique
  -> Convergence prématurée
- taux de mutation = 0.7 : nombreuses mutations donc forte variabilité génétique
  -> Convergence lente

Paramètre: probabilité de sélection par tournoi

Evolution de la distance minimale en fonction du nombre de générations

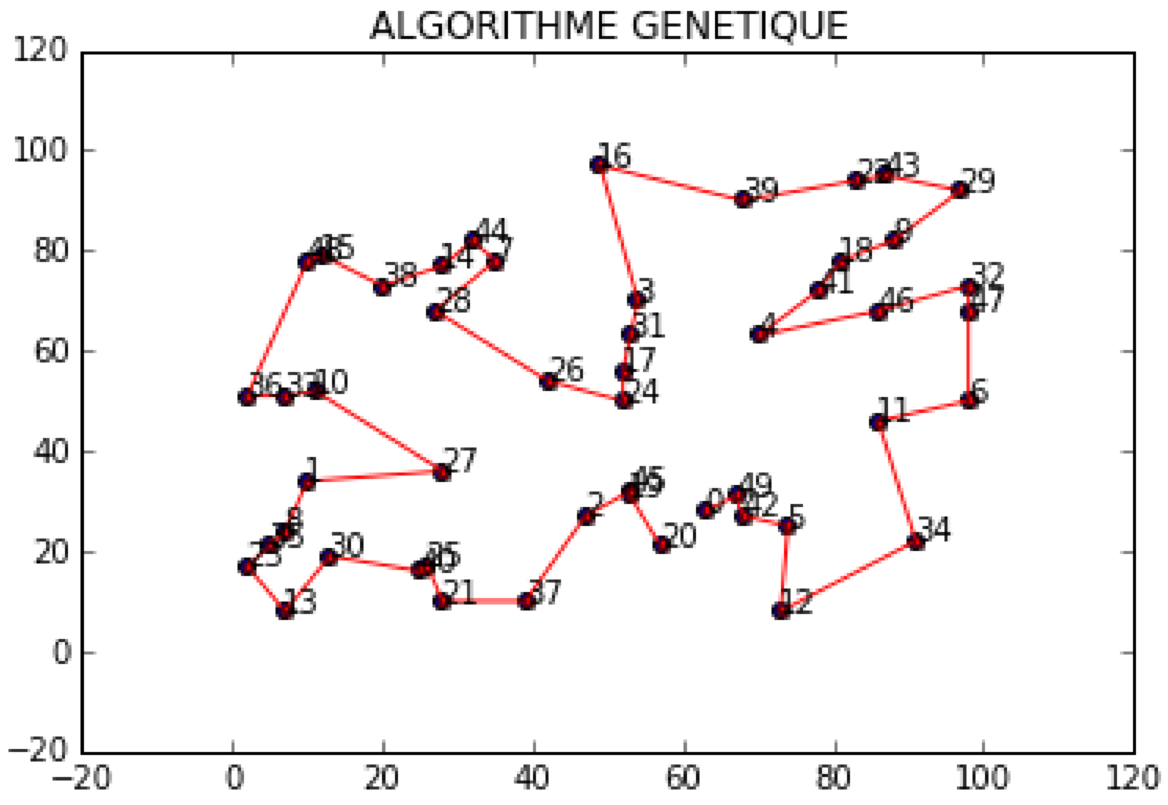


Résultat: probabilité de sélection par tournoi optimale : 0.7

Cause:

- P = 0.6: pression de sélection faible donc forte variabilité génétique
  -> Convergence lente
- P= 0.8 : pression de sélection forte donc faible variabilité génétique
  -> Convergence prématurée

Solution



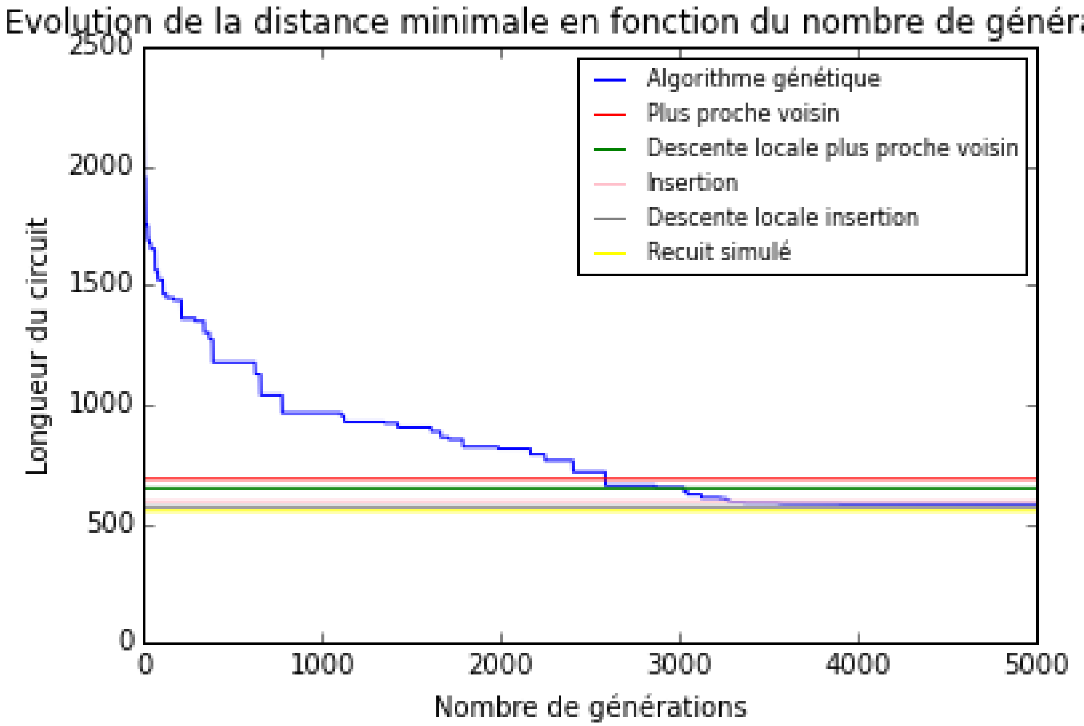
Paramètres

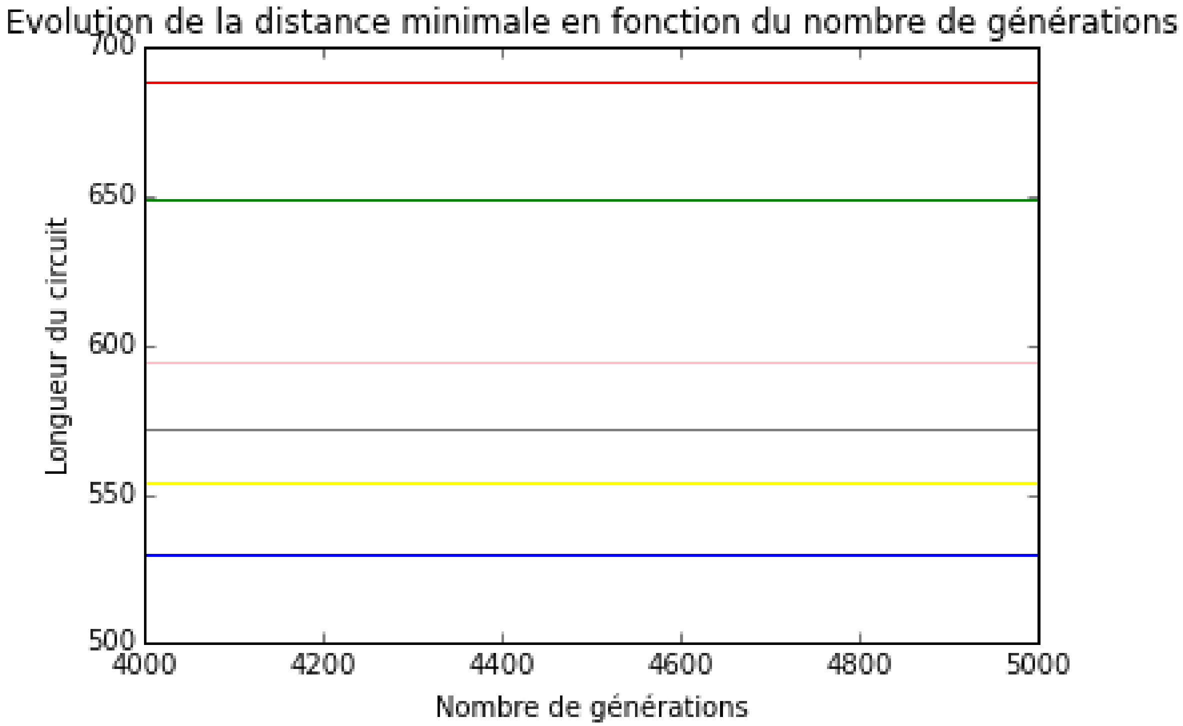
- Taux de mutation = 0.6
- Probabilité de tournoi = 0.7
- Sélection par tournoi
- Croisement simple
- Mutation bridge

Ici test pour:

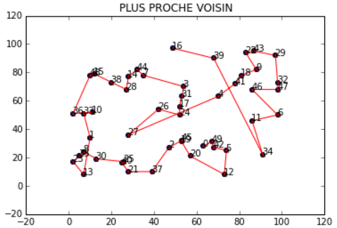
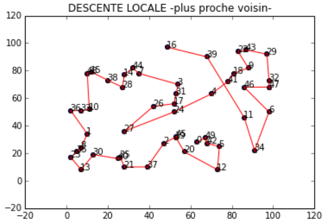
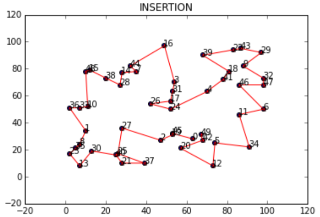
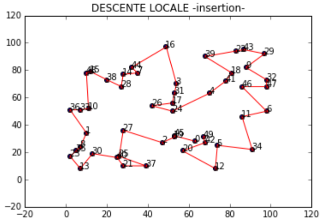
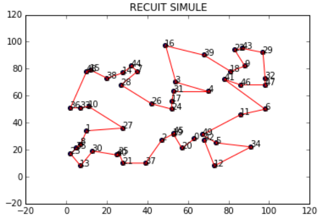
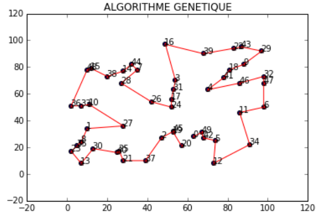
- G = 5000
- Nb d'individus = 30

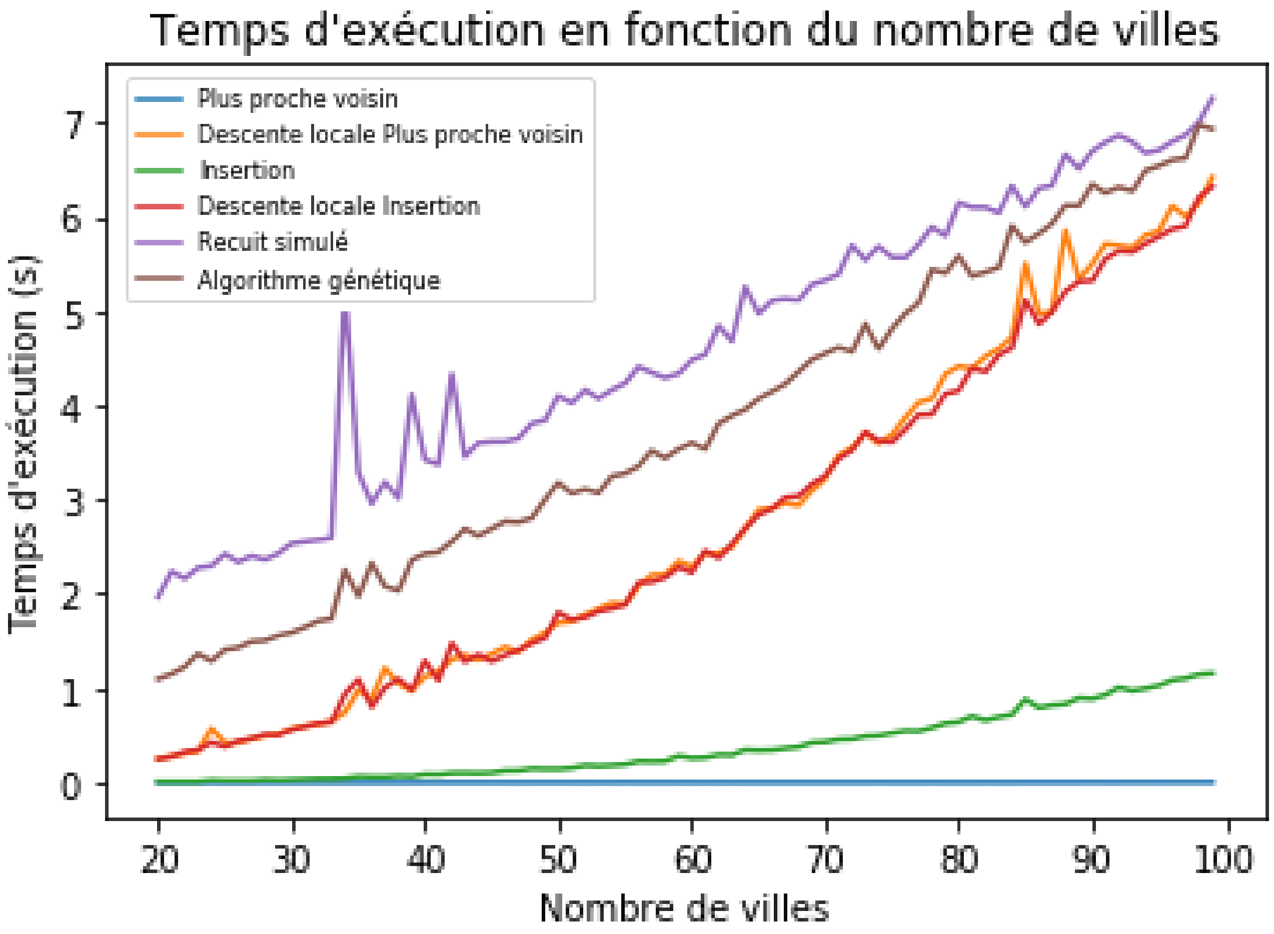
Longueur finale : 530.2 km Temps d'exécution: 2.4 s


Evolution de la distance minimale en fonction du nombre de générations
Algorithme génétique
Plus proche voisin
Descente locale plus proche voisin
Insertion
Descente locale insertion
Recuit simulé
Longueur du circuit
Nombre de générations
0
500
1000
1500
2000
2500
0
1000
2000
3000
4000
5000


Evolution de la distance minimale en fonction du nombre de générations
Longueur du circuit
Nombre de générations
500
550
600
650
700
4000
4200
4400
4600
4800
5000

3- Comparaison

Méthodes algorithmiques	Plus proche voisin	Descente locale plus proche voisin	Insertion	Descente locale insertion	Recuit simulé	Algorithme génétique
Circuit						
Durée d'exécution (s)	0.001	2	0.15	1.9	4.1	2.4
Longueur finale (km)	688.7	648.9	594.8	571.7	554.3	530.2


Temps d'exécution en fonction du nombre de villes
Plus proche voisin
Descente locale Plus proche voisin
Insertion
Descente locale Insertion
Recuit simulé
Algorithme génétique
Temps d'exécution (s)
0
1
2
3
4
5
6
7
20
30
40
50
60
70
80
90
100
Nombre de villes

Conclusion

- Fabriquer un algorithme génétique efficace est une tâche difficile : nécessité d’optimiser les paramètres
- Importance du hasard: force des algorithmes génétiques

Annexe 1: Algorithme

```
size = 50                   #nombre de villes : villes allant de 0 à size-1 (pour 10 villes, 0 à 9 sont les noms des villes)
nb = 30                     #nombre individus
G = 5000                    #nombre de generations
taux_mutation = 0.6
p = 0.7                                 #selection_tournoi: probabilité de choisir le meilleur individu parmi 2

#___________________________________________________________________________________________________________
#_____________________________________________________CREATION DES VILLES ET DE LA MATRICE D'ADJACENCE______

""" Les listes X et Y correspondent aux coordonnées des villes. La liste P correspond au couple de coordonnées.
La distance entre chaque ville est stockées dans le tableau villes, une matrice symétrique. """

X = list(np.random.randint(1,100,size))
Y = list(np.random.randint(1,100,size))
P = [X,Y]                  # liste des positions en abscisse et ordonnée de chaque ville

                           # correspond aux noms des villes (exemple: l'indice 0 correspond à la ville 0), l'indice maximal des villes est donc size-1
                           # on l'utilise cette liste pour préciser le numéro de chaque ville sur les graphes notamment avec la fonction ax.annotate()


def distance(a,b,P):                  # renvoie la distance entre deux villes
    """ a et b correspondent à deux villes """
    return ((P[0][a] - P[0][b])**2 + (P[1][a] - P[1][b])**2)**(1/2)


def generevilles(size):               # renvoie une matrice symétrique où figurent les distances entre les villes (de diagonale nulle)
    villes = np.zeros((size,size))    # np.zeros(): permet de créer une matrice où les coefficients valent tous 0

    for i in range(size):
        for k in range(0,i):

            if i != k:                # la matrice renvoyée est symétrique  et la distance d'une ville à elle-même est nulle
                d = distance(i,k,P)
                villes[i,k] = d
                villes[k,i] = d

    return villes



villes = generevilles(size)     # la matrice qui sert de base d'étude pour les différentes méthodes algorithmiques qui suivent
```

Annexe 1: Algorithme

```
#____________________________________________________________________________________________________________________
#_____________________________________________________________________GENERATION DE LA POPULATION INITIALE___________

#individu: liste du type [0,1,5,3,4,2,0] solution du pvc (liste des villes par lesquelles passent le voyageur)

#population: ensemble d'individus


def individu_aleat(villes):                  #generation aléatoire de solutions approchées (individus)
       L = [ i for i in range(1,size)]
       random.shuffle(L)
       L = [0] + L +[0]
       return L



def population(villes,nb):                   #on crée une liste de nb individus
    pop = []
    for i in range(nb):
        pop += [individu_aleat(villes)]
    return pop

#____________________________________________________________________________________________________________________
#_______________________________________________________________FONCTION D'ADAPTATION ET EVALUATION__________________


def fitness(ind):                            #fonction d'adaptation: distance totale parcourue par le voyageur
    d = 0
    for i in range(1,len(ind)):
         d += villes[ind[i-1],ind[i]]
    return d


def fittest(pop):                            # renvoie le meilleur individu dans la population (chemin le plus court)
    L = [ (fitness(pop[i]),pop[i]) for i in range(len(pop))]
    return min(L)[1]
```

Annexe 1: Algorithme

```python
#______________________________________________________________________________
#____________________________________________________SELECTION_________________

def selection_roulette(pop):
    L =  [fitness(ind) for ind in pop]
    somme_fitness =  sum(L)
    r = random.uniform(0,somme_fitness)
    s = L[0]
    i = 0
    while s < r:
        i += 1
        s += L[i]
    return pop[i]


def selection_tournoi(pop,p):
        m,n = random.randint(0,len(pop)-1),random.randint(0,len(pop)-1)     #les deux individus peuvent être les mêmes
        x = random.random()
        if fitness(pop[m]) < fitness(pop[n]):
            if x < p:                                                        #probabilité p que le meilleur individu soit sélectionné
                return pop[m]
            else:
                return pop[n]
        else:
            if x < p:
                return pop[n]
            else:
                return pop[m]
```

Annexe 1: Algorithme

```
#______________________________________________________________________________________________________________
#______________________________________________________________________CROISEMENT/RECOMBINAISON______________________________________________


def crossover_simple(pere,mere):                  #croisement simple où la cassure est situé à la moitié du parcours
    N = len(pere)//2
    fils = pere[:N]
    for k in range(N,len(mere)):
        a = mere[k]
        while a in fils and a !=0 :
              a += 1
              if a == size:
                  a = 1
        fils += [a]
    return fils

def crossover_double(pere,mere):                  #croisement double où la portion du chemin insérée est comprise entre le premier et le deuxième tiers
    N = len(pere)//3
    fils = pere[:N]
    for k in range(N,2*N):
        a = mere[k]
        while a in fils and a != 0:
              a += 1
              if a == size:
                  a = 1
        fils += [a]
    for k in range(2*N,len(pere)):
        b = pere[k]
        while b in fils and b != 0:
              b += 1
              if b == size:
                  b = 1
        fils += [b]
    return fils
```

Annexe 1: Algorithme

```
#________________________________________________________________________________
#_______________________________________MUTATION________________________________


def mutation_permutation(ind,taux_mutation):            #l'individu a une probabilité égale à taux_mutation de permuter deux villes
    for pos1 in range(1,len(ind)-1):
        if random.random() < taux_mutation:        #taux_mutation = paramètre à définir
          pos2 = random.randrange(1,len(ind)-1)
          ind[pos1],ind[pos2] = ind[pos2],ind[pos1]    #permutation des deux villes
    return ind


def mutation_bridge(ind,taux_mutation):          #l'individu a une probabilité égale à taux_mutation de former des ponts séquentiels
    i = random.randrange(1,len(ind)-10)
    k = random.randrange(i+2,len(ind)-6)
    l = k + random.randrange(0,3)
    x = random.random()
    if x < taux_mutation:
        ind = ind[:i+1] + ind[k:l+1] + ind[i+1:k] + ind[l+1:]          #pont séquentiel
    return ind
```

Annexe 1: Algorithme

```
#______________________________________________________________________________________________________________________
#_____________________________________________________________OPTIMISATION_____________________________________________


def optimisation_inversion_parcours(ind):    #optimisation de l'individu: choix du meilleur individu entre celui initial et celui dont le parcours est inverse
    for i in range(1,len(ind)-3):
        j = random.randrange(i+1,len(ind)-2)
        L = []
        S = []
        L = ind[i:j]
        L.reverse()
        S = ind[:i] + L + ind[j:]
        if fitness(S) < fitness(ind):
            ind = S
    return ind


def optimisation_inversion_parcours_v2(ind):    #même principe mais moins coûteux (on détermine seulement les variations de distance sur les arêtes modifiées)
    for i in range(1,len(ind)-3):
        j = random.randrange(i+1,len(ind)-2)
        a = villes[ind[i-1],ind[j]] + villes[ind[i],ind[j+1]]
        b = villes[ind[i-1],ind[i]] + villes[ind[j],ind[j+1]]
        if a < b:
            L = ind[i:j+1]
            L.reverse()
            ind = ind[:i] + L + ind[j+1:]
    return ind

#______________________________________________________________________________________________________________
#______________________________________________________________________REINSERTION______________________________


def reinsertion(pop,fils):                  #insertion du fils dans la population et suppression du pire individu
    L = [ (fitness(pop[i]),pop[i]) for i in range(len(pop))]
    L.append((fitness(fils),fils))
    L = [ x[1] for x in sorted(L)]
    del L[-1]
    return L
```

Annexe 1: Algorithme

```
#________________________________________________________________________________________________
#_______________________________________________________________EVOLUTION_______________________


def evolution(pop):                                                             #Evolution de la population
        pere, mere = selection_tournoi(pop,p),selection_tournoi(pop,p)                   #SELECTION des parents
        fils = crossover_simple(pere,mere)                                        #CROISEMENT des parents
        fils_mute = mutation_bridge(fils,taux_mutation)                                  #MUTATION du fils
        fils_opti = optimisation_inversion_parcours_v2(fils_mute)                          #OPTIMISATION du fils
        fils_opti = optimisation_inversion_parcours_v2(fils_mute)
        fils_opti = optimisation_inversion_parcours_v2(fils_mute)
        pop = reinsertion(pop, fils_opti)                                                #INSERTION du fils
        return pop


def generation(pop,G):                          #Amélioration de la population sur G générations et affichage de la solution finale
    for i in range(G):                                      #on teste sur G générations
        pop = evolution(pop)
    graphe(fittest(pop))
    return pop
#________________________________________________________________________________________________
#____________________________________________________________________APPLICATION_________________


pop = population(villes,nb)
generation(pop,G)
```

Annexe 2: NP-complétude

- Problèmes NP-complets: on peut vérifier l'algorithme en temps polynomial
- Mais on ne connait pas d'algorithmes qui résout le problème en temps polynomial
- En déterminer un (ou montrer qu'il n'en existe pas) permettrait de résoudre le problème de décision P = NP ou P != NP