Le Voyageur de Commerce

Année 2022 – 2023

Philippe Aumaitre, Romain Dhenry, Zoé Surelle, élèves de Terminale générale

Établissement : Lycée Raynouard, Brignoles

Enseignant·es : Denis Guicheteau, Nelly Mourau, Marc Brunet

Chercheurs : Frédéric Havet, INRIA, Thierry Champion, Université de Toulon.

1. Présentation du sujet

Un voyageur de commerce souhaite organiser sa tournée de façon optimale. Ainsi, il souhaiterait parcourir toutes ces villes en optimisant le trajet, c'est-à-dire en minimisant la distance parcourue lors de ce trajet. Pour ce faire, il n'est limité ni par le point de départ, ni par le point d'arrivée, il doit simplement passer par toutes les villes (1).
On peut donc se demander : quel est le chemin le plus court ?

2. Résultats

Pour un petit nombre de points, on pourra déterminer le plus court chemin à l'aide d'un algorithme qui teste tous les chemins possibles dans un temps factoriel.
Ensuite, pour un grand nombre de points, on a développé notre propre algorithme qui permet de calculer dans un temps polynomial un chemin court mais qui n'est pas forcément le plus court.

3. Les différentes démarches

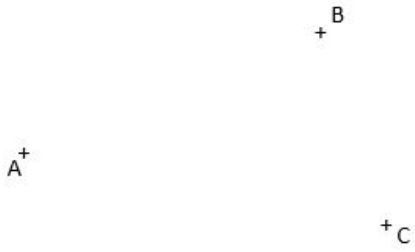
3.1. L'idée dite "naïve"

Dans un premier temps, l'idée la plus spontanée nous est venue à l'esprit : à chaque intersection, on prend le chemin le plus court sans revenir en arrière.

Pour 2 villes :

Le plus court est évidemment de les joindre par un segment

Pour 3 villes :



$$BC < AB < AC$$
Chemin minimal : AB + BC ou CB + BA

Dans le cas de 3 villes, le chemin minimum s'obtient en choisissant les deux plus courts chemins successivement.

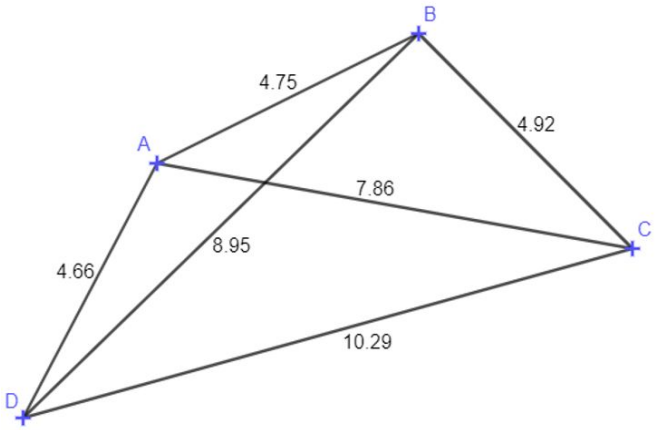
Propriété :

S'il y a trois villes A, B et C telles que AB < BC < CA,
la longueur minimale est la longueur du chemin ABC : AB + BC

Pour plus de trois villes :

On tente de prendre le chemin le plus court à chaque nœud, en évitant de retourner en arrière.

Exemple pour 4 villes :



Suivant le point de départ, nous obtenons des résultats différents :

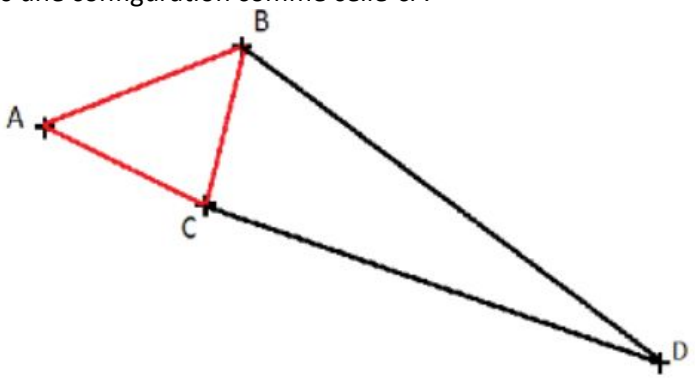
Trajet ADBC : 18,53

Trajet BACD : 19,7

Trajet CBAD : 14,33

Trajet DABC : 14,33

Autre problème, dans une configuration comme celle-ci :



L'algorithme ne parvient pas à choisir un chemin vers D car trop loin des autres (2).

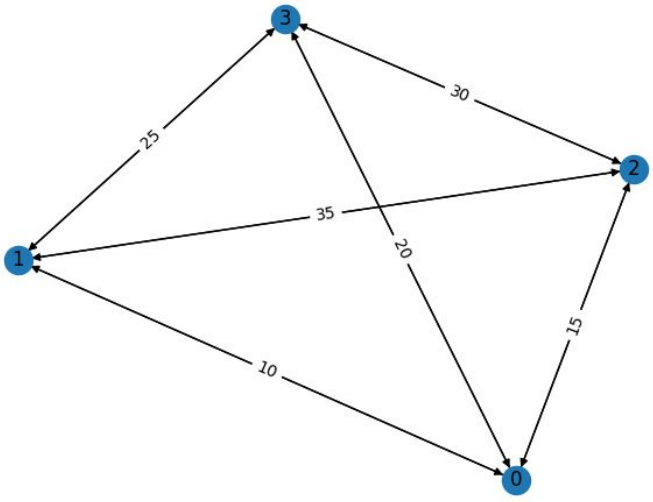
Conclusion : il faut changer d'idée.

3.2. L'utilisation de matrice :

Nous avons naturellement présenté les chemins sous forme de graphe, mais cette représentation ne nous permettait pas de faire des calculs rapidement.
Ainsi nous avons changé et nous sommes passés à une représentation sous forme de matrice (3).
Par exemple :

[0, 10, 15, 20],
[10, 0, 35, 25],
[15, 35, 0, 30],
[20, 25, 30, 0]

correspondait à ce graphe ci :

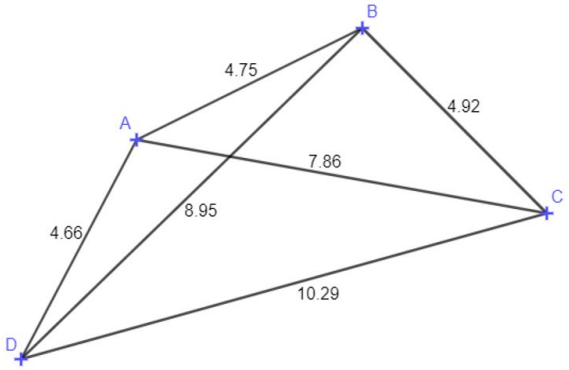


De plus, nous avons aussi créé un programme qui prend en paramètre des coordonnées de points et qui renvoie la matrice correspondant en calculant les distances entre chaque point.

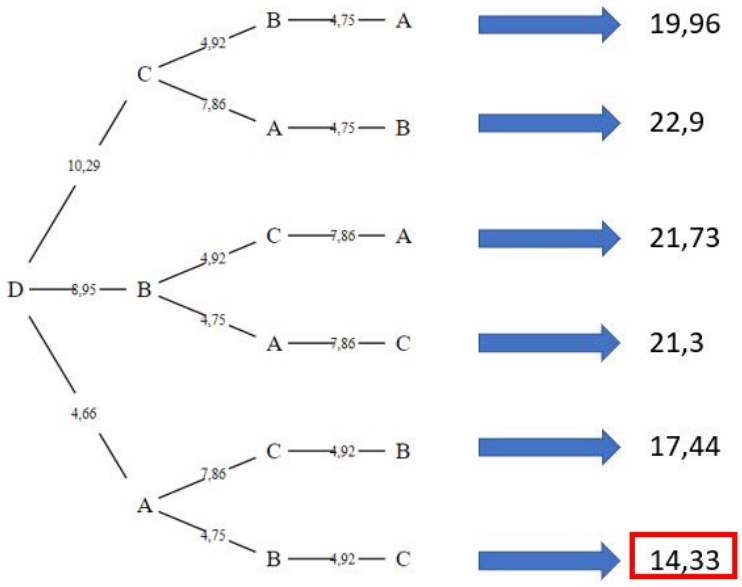
```
import numpy as np
#fonction qui prend 7 points et construit la matrice de tous les chemins
possibles
def graph_routes():
  list = [0] * 7
  dist = np.zeros((7, 7))
  for i in range(7):
    x = input('Points : ').split(' ') (4)
    list[i] = x
  for i in range(7):
    a = list[i]
    for j in range(7):
      b = list[j]
      dist[i][j] = round(sqrt((int(a[0])-int(b[0]))**2+(int(a[1])-
int(b[1]))**2),2)
  return dist
```

3.2. L'algorithme de force brute.

Dans un second temps, une idée plutôt logique s'est imposée à nous : on calcule tous les chemins possibles. Cette démarche est réalisable à la main à l'aide d'un arbre pondéré.



Exemple d'arbre pour un départ de D.



Pour 4 points, il a fallu faire 24 calculs pour trouver tous les chemins possibles.
Pour n points, il y a n factorielle possibilités.

QU'EST-CE QUE LA FONCTION "FACTORIELLE" ?

La fonction factorielle de n (notée "$n!$"), correspond au produit des nombres de 1 à n (par exemple $4! = 1 \times 2 \times 3 \times 4$). Cette notion est utilisée dans la combinatoire, plus précisément dans le cas des permutations.
Une permutation, c'est lorsque j'ai par exemple 10 livres dans une bibliothèque, et que je souhaite les ranger tous aléatoirement : j'aurai alors $10!$ rangements de livres différents possibles.

Cela signifie que le nombre de calculs peut vite devenir extrêmement grand, et donc long à calculer. Pour vous donner un ordre d'idée, l'ordinateur le plus puissant du monde peut effectuer environ 10^{15} calculs à la seconde, or $20\ !$ vaut environ $2{,}4 \cdot 10^{18}$, et il y a bien plus de 20 villes rien qu'en France.

Conclusion : Il faut changer d'idée. Mais pour de petits nombres, nous allons nous servir de cet algorithme comme base pour comparer nos futures idées.

Programme Python de force brute : (5)

```
import itertools
import numpy as np
def tsp(graph):¬
  # obtenir tous les chemins possibles à  partir du noeud de départ¬
  paths = list(itertools.permutations(range(len(graph))))

  # initialiser la distance minimale à  l'infini¬
  min_distance = float('inf')
  min_path = []
  # pour chaque chemin
  for path in paths:
    # initialiser la distance totale à  0
    total_distance = 0

    # pour chaque noeud dans le chemin
    for i in range(len(path) - 1):

      if graph[path[i]][path[i+1]]==0:
          total_distance=float('inf')
          break
      # ajouter la distance entre le noeud actuel et le prochain¬
      total_distance += graph[path[i]][path[i+1]]

    # si la distance totale est inférieure à  la distance minimale actuelle¬
    if total_distance < min_distance:
      # mettre à  jour la distance minimale et le chemin minimal¬
      min_distance = total_distance
      min_path = path

# renvoyer la distance minimale et le chemin minimal¬
  return min_distance, min_path
```

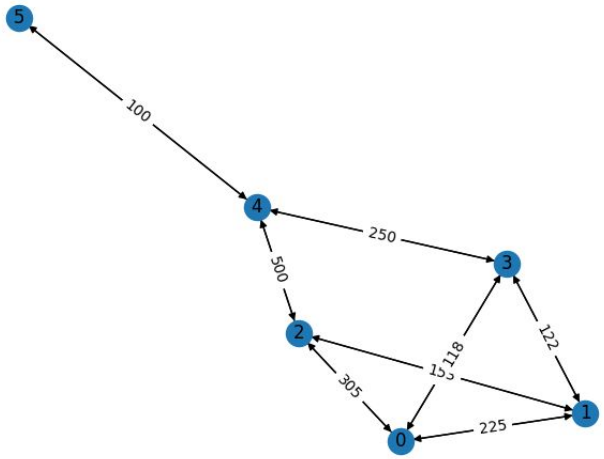
```
# exemple d'utilisation
graph = [[0, 10, 15, 20],
         [10, 0, 35, 25],
         [15, 35, 0, 30],
         [20, 25, 30, 0]]

min_distance, min_path = tsp(graph, 0)
print(min_distance)  # 50
print(min_path)  # (2, 0, 1, 3)
```

3.3. L'idée finale

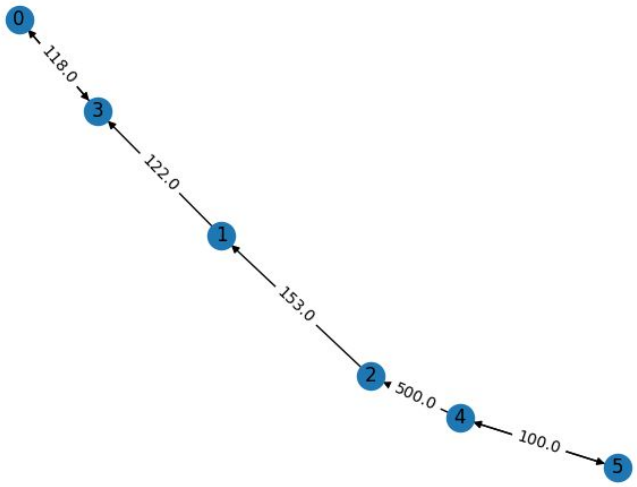
Enfin, l'idée sur laquelle nous nous sommes concentrés correspond à la démarche suivante :

- Nous relions des grappes de points les plus proches (avec la méthode naïve).
- Ensuite nous relions les extrémités des grappes. (6)

Par exemple, une configuration comme celle-ci :



Donnera une grappe comme celle-ci :



Cependant, cette idée nous donne un chemin court, mais pas forcément le plus court.

Ainsi, il faut vérifier si notre solution est satisfaisante, nous utilisons en premier lieu l'algorithme de force brute pour comparer notre solution avec la solution optimale.

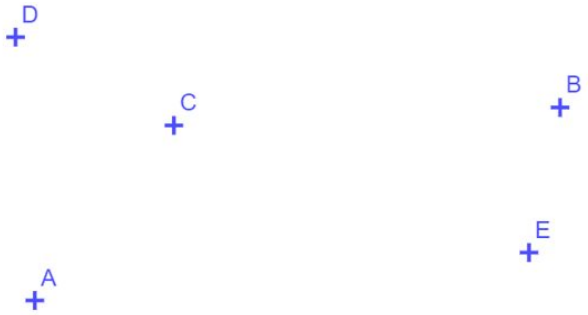
Malheureusement, si le nombre de points devient grand, il est difficile de vérifier si notre solution donne bien un chemin suffisamment court.

De là, nous avons décidé de mettre en place une borne inférieure, qui nous permet d'avoir une idée de la longueur finale du chemin que nous renvoie notre algorithme.

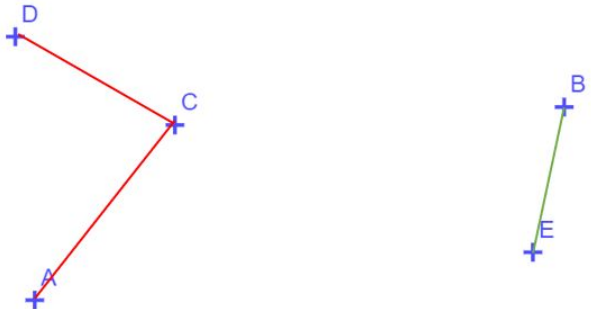
LA BORNE INFÉRIEURE : LE CALCUL (7)

Comme dit ci-dessus, on souhaite calculer une valeur basse pouvant nous donner une idée de la longueur minimale du chemin le plus court.
Ainsi, pour relier nos n points, il faut $(n-1)$ segments.
On va prendre pour chaque point, la longueur la plus petite, on additionne ces n longueurs. Puis on multiplie par $\frac{n-1}{n}$ pour obtenir une valeur approchée de la longueur des $(n-1)$ segments.

Fonctionnement de l'algorithme sur un exemple :

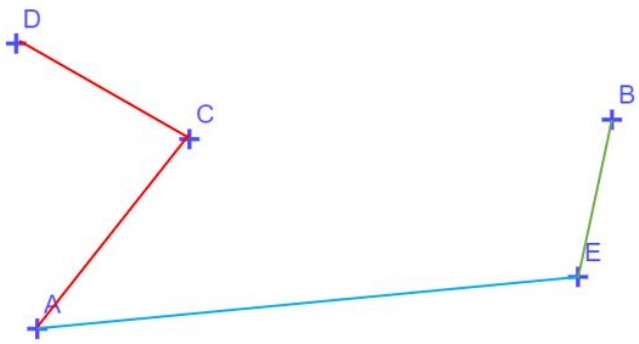


1ère étape : on relie les points les plus proches en utilisant la matrice.

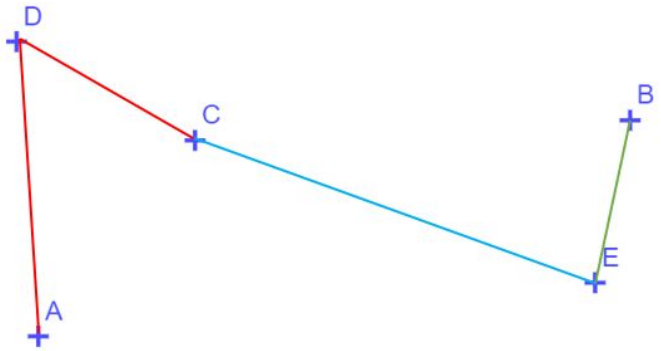


Ainsi, A est relié à C, B est relié à E, C est relié à D, D est relié à C et E est relié à B.

2ème étape : on regarde parmi les extrémités des grappes, le chemin le plus court pour les relier



On peut voir que dans ce cas, le chemin n'est pas optimal car le chemin suivant est meilleur : (8)



C'est le chemin que nous obtenons par force-brute.

Néanmoins, notre algorithme nécessite dans le meilleur des cas n calculs (9). Nous n'avons pas réussi à déterminer le nombre de calculs pour le pire des cas.

Voici un lien vers notre programme : https://trinket.io/python3/507ab39819 (10)

4. Conclusion

Le problème du voyageur de commerce est en soi facile à résoudre, mais c'est le temps de résolution qui pose problème.

En effet, la raison pour laquelle le voyageur de commerce fait partie des problèmes du millénaire, c'est parce qu'on ne sait pas si on peut le résoudre dans un temps polynomial. Pour le dire autrement, le temps de calcul augmente d'une manière bien trop considérable lorsque qu'on augmente le nombre de points

QU'EST-CE QU'UN PROBLEME DU MILLENAIRE ?

En l'an 2000, 7 problèmes ont été introduits par l'institut de mathématiques Clay et dont la résolution apporte la modique somme d'un million de dollars. Ce sont tous des problèmes extrêmement difficiles à résoudre de manière précise et nécessitant des techniques mathématiques très avancées pour même s'approcher d'une solution à peine précise. Pour le moment, seule la conjecture de Poincaré a été résolue par Grigori Perelman, un mathématicien russe.

5. Appendice : le programme Python

```
import numpy as np
#calcule un minimum du graphe et construit la matrice qui relie avec les minimums
def mini(graph):
  #on récupère la taille de la matrice carrée
  taille=len(graph)
  #on construit une nouvelle matrice carrée avec que des 0
  graph1= np.zeros((taille,taille))
  #on crée un tableau des connexions déjà  faites avec deux colonnes
  #la première avec un antécédent et la suivante avec le précedent(unique)
  #on met -1 si pas de connexion
  co=np.zeros((taille,2))-1
  #initialisation du minimum total
  m=0
  #on initialise les indices des minima
  imin=0
  jmin=0
  #on va parcourir chaque ligne
  for i in range(taille):
    #on cherche le minimum donc on commence par comparer à l'infini
    mi=float('inf')
    #on parcourt la ligne
    for j in range (taille):
  #si le chemin ne vaut pas 0 et est plus petit que le min en cours, on le garde
      if graph[i][j]!=0 and graph[i][j]<mi:
        mi=graph[i][j]
        imin=i
        jmin=j
    #on rajoute ce min au total
    m=m+mi
    #on trace le chemin dans la matrice
    graph1[imin][jmin]=mi
    #on rajoute la connexion dans le tableau de connexion
    #on relie imin à  jmin donc un antécedent de jmin est imin
    #et le précédent de imin est jmin
    co[imin][1]=jmin
    if co[jmin][0]==-1:
      co[jmin][0]=imin
    elif co[jmin][1]!=imin:
      co[jmin][0]=imin
    else :
      a=co[jmin][0]
      co[jmin][0]=co[jmin][1]
      co[jmin][1]=a
  return round(m*(taille-1)/taille,2),graph1,co

#fonction qui établit les connexions manquantes
def connexion(graph,graph1,co):
  #on enlève les chemins déjà pris
  g=graph-graph1
  #taille de la matrice
  taille=len(graph)
  #on cherche un premier chemin
  chemin=[]
  #on cherche si il y a un sommet sans antécédent
  for i in range(taille):
```

```
    if co[i][0]==-1 and len(chemin)==0:
      chemin.append(i)
  i=0
  #on regarde si on a trouvé ce sommet
  if len(chemin)!=0:
    #si c'est le cas, on va suivre les précédents jusqu'à  un retour
    #c'est à  dire précédent=antécédent
    j=chemin[i]
    while (int(co[j][1]) not in chemin) or (int(co[j][0]) not in chemin):
      if co[j][0]==-1:
        co[j][0]=co[j][1]
      if int(co[j][1]) not in chemin:
        chemin.append(int(co[j][1]))
      elif int(co[j][0]) not in chemin:
        chemin.append(int(co[j][0]))
      else:
        break
    i=i+1
    j=chemin[i]
    k=i
  #si on n'a pas de point de départ (que des boucles)
  else:
    #on va vérifier que tous les sommets ne soient pas reliés
    #on cherche les doubles
    i=0
    while i <(taille) and co[i][1]!=co[i][0]:
        i=i+1
    #si on n'a pas trouvé de double on repart de 0
    if i==taille:
      i=0
    #on va suivre le chemin en prenant l'antécedent ou le précédent
    #car le chemin peut aller dans les deux sens
    chemin.append(i)
    chemin.append(int(co[i][1]))
    k=1
    j=chemin[k]
     while co[j][1]!=co[j][0] and (int(co[j][1]) not in chemin or int(co[j][0]
not in chemin)):
      if co[j][0]==chemin[k-1]:
        chemin.append(int(co[j][1]))
      else:
        chemin.append(int(co[j][0]))
      k=k+1
      j=chemin[k]

  #si notre chemin correspond à  la taille du graph alors tout est relié
  if k==len(graph)-1:
    print("un chemin est ",chemin)
    lon=0
    for i in range(taille-1):
      lon=lon+graph[chemin[i]][chemin[i+1]]
    print("longueur=",lon)
    return True

  #si on arrive ici, le chemin n'est pas complet donc
  #on va regarder les extremités du chemin avec les autres morceaux
  lien=float('inf')
```

```
  ilien=-1
  #on prend notre début de chemin et la fin de notre chemin
  debut=chemin[0]
  fin=chemin[-1]
  #on va chercher parmi les arêtes non explorées, celles qui sont
  #les plus courtes et qui relient notre début ou notre fin à un
  #élément qui n'est pas du chemin
  for i in range(taille):
    #on rappelle que 0 correspond à une absence d'arête
    if (i not in chemin) and (g[debut][i]<lien)and (g[debut][i]!=0):
      ilien=i
      lien=g[debut][i]
      #on met une variable h pour savoir entre le début et la fin qui est le
      #mieux raccordé
      h=0
  #si le début n'est pas lié on regarde la fin
  for i in range(taille):
    if (i not in chemin) and (g[fin][i]<lien)and (g[fin][i]!=0):
      ilien=i
      lien=g[fin][i]
      h=1 # si c'est la fin qu'il vaut mieux relier
  #si la fin est le début ne sont pas liés, il va falloir réfléchir
  #mais pour l'instant je mets erreur !
  if ilien==-1:
    print("erreur")
    return False
  elif h==1:
  #si on a trouvé un lien avec la fin, on le matérialise en changeant les
  #connexions et le graph1
    graph1[fin][ilien]=lien
    co[fin][1]=ilien
    co[ilien][0]=fin
  else:
  #même chose si le lien est au début
    graph1[ilien][debut]=lien
    co[debut][0]=ilien
    co[ilien][1]=debut
#une fois le lien fait, on relance la machine pour completer le chemin
connexion(graph,graph1,co)

# exemple d'utilisation
graph1 = np.array(
        [[0,225,305,118,0,0],
         [225,0,153,122,0,0],
         [305,153,0,0,500,0],
         [118,122,0,0,250,0],
         [0,0,500,250,0,100],
         [0,0,0,0,100,0]
])
graph=graph1
min_distance, min_path = tsp(graph)
print(min_distance)
print(min_path)
m,g,c=mini(graph)
print("mini theorique:",m)
connexion(graph,g,c)
```

Notes d'édition

(1) Dans cet article on ne prend pas en compte le trajet de retour du voyageur de commerce à son point de départ.

(2) Ici, l'algorithme trouve un chemin à partir de A ou de D, mais à partir de B il va en C puis en A où il n'y a pas de chemin vers D sans repasser par B ou C, et de même à partir de C il va en B puis en A où il est bloqué.

(3) La matrice associée comporte autant de lignes et de colonnes qu'il y a de villes, et le j-ème terme de la ligne i est la longueur du chemin direct de la ville i à la ville j si $i \neq j$ et s'il en existe un, et 0 sinon.

(4) L'instruction `x=input('Points : ').split(' ')` demande à entrer des deux coordonnées d'un point séparées par un espace, puis les retourne sous forme d'une liste `(x[0], x[1])`.

(5) Dans le programme, `graph` est en fait la matrice des distances définie précédemment. `list(itertools.permutations(range(n)))` produit la liste de toutes les permutations des entiers de 0 à n-1, c'est-à-dire ici de tous les chemins à examiner.

Il aurait été intéressant d'avoir plusieurs exemples d'utilisation avec leur temps d'exécution, notamment pour voir comment ce temps augmente en fonction de la taille des données.

(6) Plus précisément le programme (présenté en appendice) comporte deux procédures :

– `mini` : avec la méthode naïve, on choisit pour chaque point le chemin le plus court qui la joint à un autre point.

– `connexion` : ensuite, en suivant ces chemins à partir d'un point donné, tant qu'on ne revient pas en arrière, on obtient une "grappe". Si elle contient tous les points, on affiche le résultat et on arrête. Sinon, on cherche à relier l'une de ses extrémités à un point extérieur à la grappe, et on relance la procédure.

Mais dans le cas où aucune de ces extrémités n'est reliée à un point extérieur le programme affiche "erreur" et s'arrête. Or ce cas peut se produire, et il faudrait plutôt faire cette recherche pour tous les points de la grappe. Il est aussi possible qu'aucun point de la grappe ne soit relié à un point extérieur – alors le problème n'a pas de solution et il faudrait l'afficher.

(7) Cette borne est affichée à l'exécution du programme. Pour être tout à fait correct, il ne s'agit pas d'une borne inférieure mais seulement d'un *minorant* de la longueur d'un chemin joignant tous les points. Pour justifier le calcul fait ici, un tel chemin est composé de n–1 chemins allant d'une ville à une autre. On peut choisir l'une des deux extrémités pour chacun de ces n–1 chemins, sans prendre deux fois la même ville, et alors la longueur totale est supérieure ou égale à la somme des plus petites longueurs des chemins partant de chacune de ces n–1 villes. De plus on voit facilement qu'on peut choisir arbitrairement la ville exclue, donc on peut retirer n'importe quel terme de la somme.

(8) On voit sur cet exemple qu'il n'est pas satisfaisant de ne chercher à relier que les extrémités des grappes.

(9) Ceci n'est pas exact, car la première procédure comporte deux boucles imbriquées de longueur n et nécessite un nombre de comparaisons de l'ordre de n^2. Le nombre total de calculs est de cet ordre de grandeur – très largement inférieur à $n!$ – lorsqu'on a un chemin complet après exécution de `mini` sans avoir à chercher d'autres connexions.

(10) Ce programme est reproduit en appendice (ajouté à l'édition).