Épreuve de TIPE :
La ville et ses réseaux de transport

Présentation de **Ryan BOUCHOU**

travail réalisé avec Baptiste Arrix-Pouget

30713

Problématisation

La ville et ses enjeux :

$\rightarrow$ Densité de population et déplacements pendulaires croissants

$\rightarrow$ Gestions et intrication des réseaux / moyens de déplacement

Problématisation

La ville et ses enjeux :

$\rightarrow$ Densité de population et déplacements pendulaires croissants

$\rightarrow$ Gestions et intrication des réseaux / moyens de déplacement

Problématique : Comment placer des stations de transports en commun en ville.

Définition du problème

entrées : Un graphe non orienté et pondéré G=($\mathcal{V}$,$\mathcal{E}$,c)
où $c : \mathcal{E} \to \mathbb{R}^+$
Une famille $\mathcal{I} \subset \mathcal{E} \times \mathbb{R}^+$
Une famille $\mathcal{P} \subset \mathbb{R}^+$
$N \in \mathbb{N}^+$

sortie : $\mathcal{S} = (S_j)_{j \in [0..N]}$ *minimisant f*

où :

Définition du problème

entrées : Un graphe non orienté et pondéré G=($\mathcal{V}$,$\mathcal{E}$,c)
où $c : \mathcal{E} \to \mathbb{R}^+$
Une famille $\mathcal{I} \subset \mathcal{E} \times \mathbb{R}^+$
Une famille $\mathcal{P} \subset \mathbb{R}^+$
$N \in \mathbb{N}^+$

sortie : $\mathcal{S} = (S_j)_{j\in[0..N]}$ *minimisant f*

où :

→ L'ensemble des sommets $\mathcal{V}$ représente les intersections des rues.

Définition du problème

entrées : Un graphe non orienté et pondéré G=($\mathcal{V}$,$\mathcal{E}$,c)
où $c : \mathcal{E} \to \mathbb{R}^+$
Une famille $\mathcal{I} \subset \mathcal{E} \times \mathbb{R}^+$
Une famille $\mathcal{P} \subset \mathbb{R}^+$
$N \in \mathbb{N}^+$

sortie : $\mathcal{S} = (S_j)_{j\in[0..N]}$ *minimisant f*

où :

→ L'ensemble des sommets $\mathcal{V}$ représente les intersections des rues.

→ L'ensemble des arêtes $\mathcal{E}$ représente les rues, potentiellement segmentées.

Définition du problème

entrées : Un graphe non orienté et pondéré G=($\mathcal{V}$,$\mathcal{E}$,c)
où $c : \mathcal{E} \to \mathbb{R}^+$
Une famille $\mathcal{I} \subset \mathcal{E} \times \mathbb{R}^+$
Une famille $\mathcal{P} \subset \mathbb{R}^+$
$N \in \mathbb{N}^+$

sortie : $\mathcal{S} = (S_j)_{j\in[0..N]}$ *minimisant f*

où :

→ L'ensemble des sommets $\mathcal{V}$ représente les intersections des rues.

→ L'ensemble des arêtes $\mathcal{E}$ représente les rues, potentiellement segmentées.

→ La famille $\mathcal{I}$ représente la répartition d'immeubles le long des rues.

- Chaque élément de $\mathcal{I}$ est au format $((u, v), d)$

Définition du problème

entrées : Un graphe non orienté et pondéré G=($\mathcal{V}$,$\mathcal{E}$,c)
où $c : \mathcal{E} \to \mathbb{R}^+$
Une famille $\mathcal{I} \subset \mathcal{E} \times \mathbb{R}^+$
Une famille $\mathcal{P} \subset \mathbb{R}^+$
$N \in \mathbb{N}^+$

sortie : $\mathcal{S} = (S_j)_{j \in [0..N]}$ *minimisant f*

où :

- $\rightarrow$ L'ensemble des sommets $\mathcal{V}$ représente les intersections des rues.
- $\rightarrow$ L'ensemble des arêtes $\mathcal{E}$ représente les rues, potentiellement segmentées.
- $\rightarrow$ La famille $\mathcal{I}$ représente la répartition d'immeubles le long des rues.
  - Chaque élément de $\mathcal{I}$ est au format $((u, v), d)$
- $\rightarrow$ La famille $\mathcal{P}$ indexe le poids de chaque immeuble (ie le nombre d'habitants).
- $\rightarrow$ N le nombre de stations à placer.

Fonction objectif

On considère :

- $\rightarrow$ Une instance $(G, \mathcal{I} = (i_k)_{k \in [0..h]}, \mathcal{P} = (p_k)_{k \in [0..h]}, N), h = |\mathcal{I}|$
- $\rightarrow$ **d** la distance dans le graphe G
- $\rightarrow$ Ue famille de N stations $\mathcal{S} = (S_j)_{j \in [0..N]}$

Fonction objectif

On considère :

- $\rightarrow$ Une instance $(G, \mathcal{I} = (i_k)_{k \in [0..h]}, \mathcal{P} = (p_k)_{k \in [0..h]}, N), h = |\mathcal{I}|$
- $\rightarrow$ **d** la distance dans le graphe G
- $\rightarrow$ Ue famille de N stations $\mathcal{S} = (S_j)_{j \in [0..N]}$

On définit la fonction objectif f par :

$$\boxed{f(S) = \sum_{k=0}^{h} \min_{j \in [1..N]} (p_k * \mathbf{d}(i_k, S_j))}$$

Recuit simulé

Algorithme 1: Recuit-simulé

Entrée: Un graphe pondéré $G = (V, E, c)$ où $c : E \to \mathbb{R}^+$, $I \subset E$
Sortie: $I^* \in argmin(f)$

$T := T_0$
$I_c := I$
tant que $T > T_{arret}$ **faire**
    I_{temp} :=Générer une nouvelle solution
    $\Delta = f(I_c) - f(I_{temp})$
    if $\Delta > 0$ **then**
        $I_c = I_{temp}$
    else
        $u := \mathcal{U}(0, 1)$
        if $u > e^{\frac{\Delta}{T_c}}$ **then**
            $I_c = I_{temp}$
    $T_c = T_c * 0.999$
retourner I_c

Algorithme génétique

Algorithme 2: Algorithme génétique

Entrée: Un graphe pondéré $G = (V, E, c)$ où $c : E \to \mathbb{R}^+$, $N \in \mathbb{N}$
Sortie: $I^* \in argmin(f)$

Pop :=Génération d'un ensemble de solutions
$\mathcal{H} := \emptyset$
tant que Condition d'arrêt **faire**
  Choix de deux parents à partir de Pop
  Croisements
  Mutations
  $\mathcal{H} \leftarrow$Résultante des opérations précédentes
retourner $\text{argmin}_{X \in \mathcal{H}} f(X)$

- **Obtention d'un filaire de voirie**

Figure – Modèle cartographique

```
{"dtasetid":"filaire-de-voirie",
"recordid":"6ff917dc1ac",
"fields":{
    "longueur":146.230773891848,
    "fromright":0,
    "codsti":313550000047,
    "geo_shape":{
        "coordinates":[
        [[1.571729386142422,
            3.612140358497975],
            [1.570169867735635,
                43.612809992835835]]
        ],
        "type":"MultiLineString"
    },
    "code_insee":31355,
    "toright":0,
    "nrivoli":"3103550030",
    "motdir":"AV DES PYRENEES",
    "toleft":0,}}
```

1 - **Toulouse Métropole, 2011**

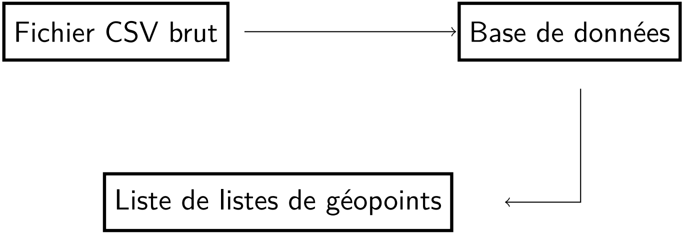
Pré-traitement

- Première étape du processus :

Fichier CSV brut → Base de données → Liste de listes de géopoints

Pré-traitement

- Première étape du processus :



→ Méconnaissance des liens entre les différentes rues

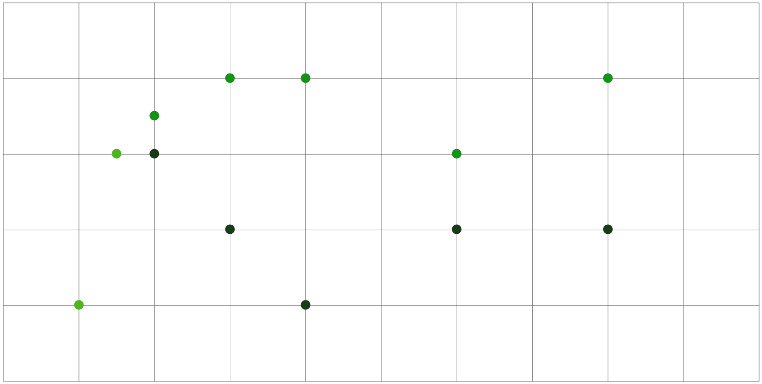
→ Coordonnées des géopoints toutes différentes

→ Données désordonnées

Étape 1 : encodage des rues

Considérons une liste de géopoints

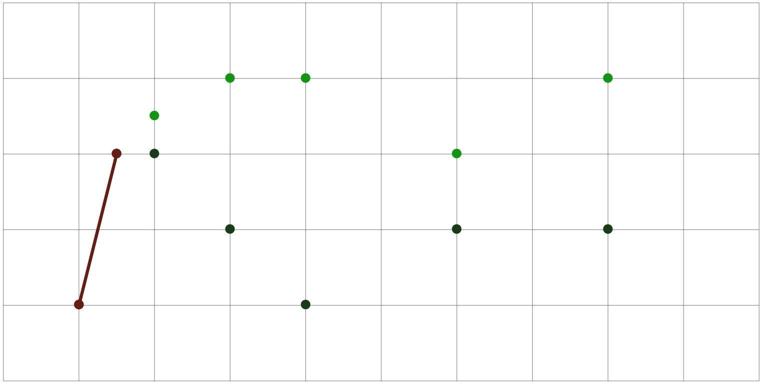
$$\Big[\big[(1,1),(1.5,3)\big],\big[(2,3),(3,2),(6,2),(8,2)\big],\big[(2,3.5),(2,4),(4,4),(8,4)\big]\Big]$$



Étape 1 : encodage des rues

Création des arêtes pour la première rue

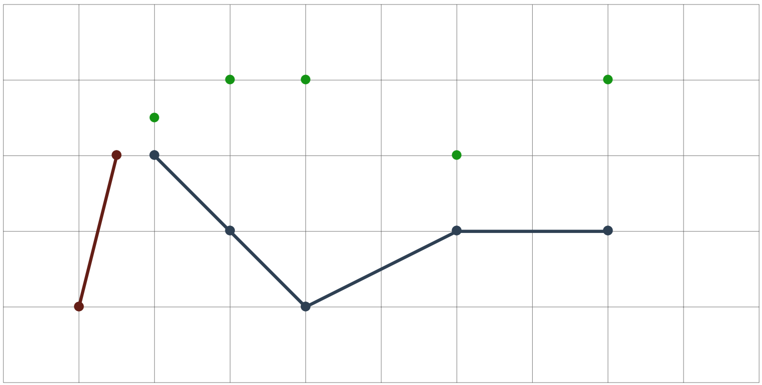
$$\Big[\big[(1,1),(1.5,2.5)\big],\big[(2,3),(3,2),(6,2),(8,2)\big],\big[(2,3.5),(3,4),(4,4),(8,4)\big]\Big]$$



Étape 1 : encodage des rues

Création des arêtes pour la seconde rue

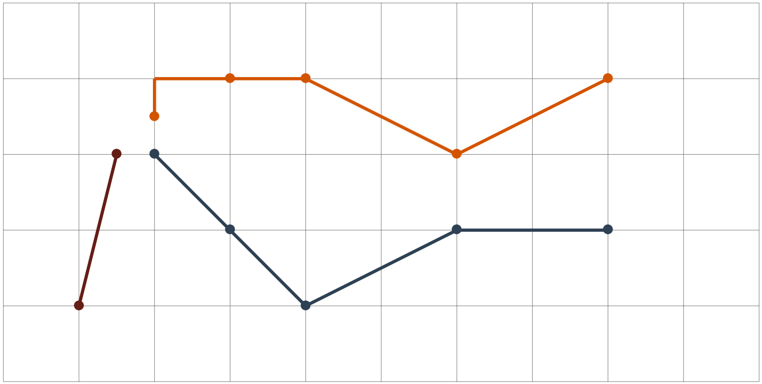
$$\Big[\big[(1,1),(1.5,2.5)\big],\big[(2,3),(3,2),(6,2),(8,2)\big],\big[(2,3.5),(3,4),(4,4),(8,4)\big]\Big]$$



Étape 1 : encodage des rues

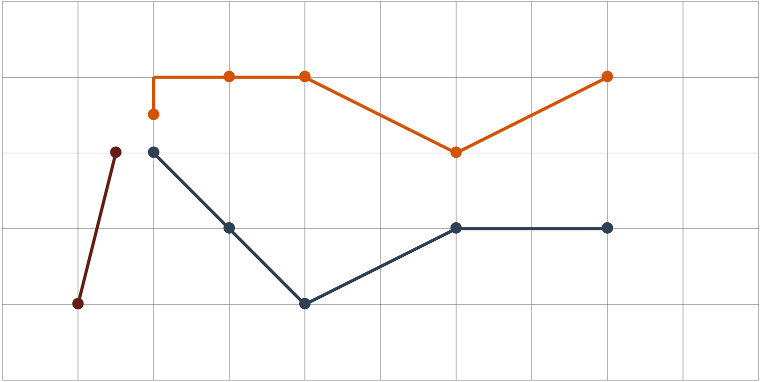
Création des arêtes pour la troisième rue

$$\Big[\big[(1,1),(1.5,2.5)\big],\big[(2,3),(3,2),(6,2),(8,2)\big],\big[(2,3.5),(3,4),(4,4),(8,4)\big]\Big]$$



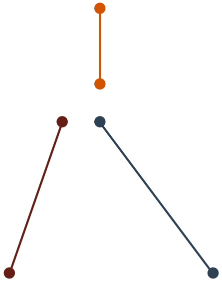
Étape 1 : encodage des rues

Objectif : passer d'un graphe éclaté à connexe



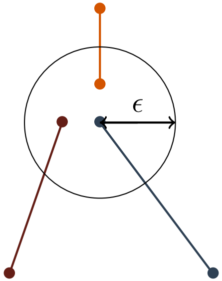
Étape 2 : connexifier le graphe

Réalisation des jonctions



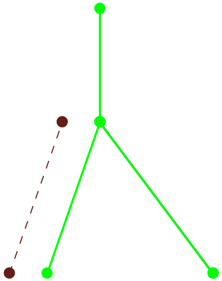
Étape 2 : connexifier le graphe

Réalisation des jonctions



Étape 2 : connexifier le graphe

Réalisation des jonctions



Première approche

Algorithme 3: Connexification

Entrée: $G = (V, E)$ $(n = |V|)$
T un tableau indicé par $[1..n]$
$\epsilon > 0$
Sortie: $G' = (V, E')$

```
i,j=1
G' ← G
pour i ≤ n faire
    pour i<j ≤ n faire
        if D(T[i], T[j]) < ϵ then
            Fusionner i et j dans G'
retourner G'
```

où :

- On considère $\mathcal{D}$ la distance euclidienne à l'aide de la formule de Harvesine :

$$\mathcal{D} = \sqrt{\sin \frac{\phi_2 - \phi_1}{2}^2 + \cos \phi_2 * \cos \phi_1 * \sin \frac{\lambda_2 - \lambda_1}{2}^2}$$

Mais...

→ Complexité en $\mathcal{O}(n^2)$

→ Réalise des comparaisons inutiles

Seconde approche

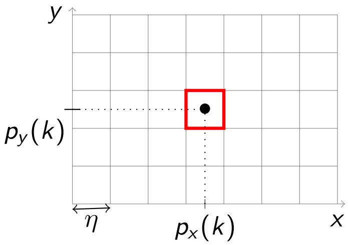
Algorithme 4: Connexification

Entrée:
$G = (V, E)$ $(n = |V|)$
T un tableau indicé par $[1..n]$
$\eta, \epsilon > 0$
Sortie: $G' = (V, E')$

Partitionnement spatial de V selon η
[...]

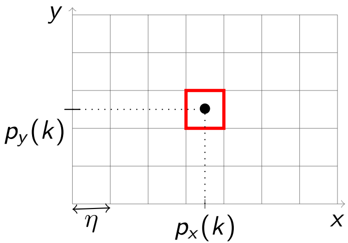
Le partitionnement

- → On détermine $(longmin, latmin)$
- → On se place dans (O, e_x, e_y) où $O = (longmin, latmin)$
- → On définit $\forall k \in V, p_x(k) = \mathcal{D}([long_k, latmin], T[k])$ et similairement p_y



Le partitionnement

- → On détermine $(longmin, latmin)$
- → On se place dans (O, e_x, e_y) où $O = (longmin, latmin)$
- → On définit $\forall k \in V, p_x(k) = \mathcal{D}([long_k, latmin], T[k])$ et similairement p_y



- → On détermine (α, β) tels que $\begin{cases} \eta * \alpha \leq p_x(k) < \eta * (\alpha + 1) \\ \eta * \beta \leq p_y(k) < \eta * (\beta + 1) \end{cases}$

Seconde approche

Algorithme 5: Connexification

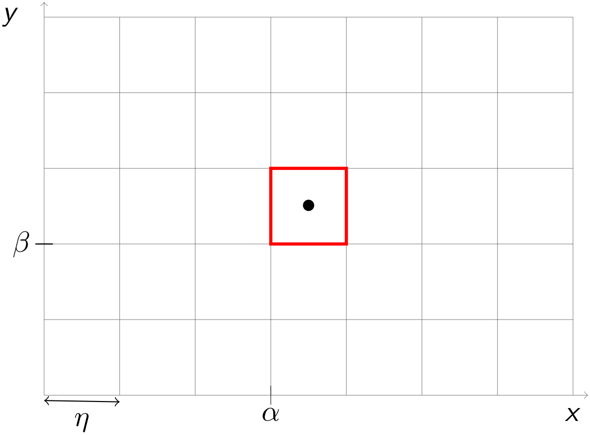
Entrée:
$G = (V, E)$ $(n = |V|)$
T un tableau indicé par $[1..n]$
$\eta, \epsilon > 0$
Sortie: $G' = (V, E')$

Partitionnement spatial de V selon η
$\mathcal{O} = V$
pour $k \in V \cap \mathcal{O}$ **faire**
    $\mathcal{F} \leftarrow \emptyset$
    pour $v \in \mathcal{P}(k) \cap \mathcal{O}$ **faire**
        $\mathcal{H} = \mathcal{H} \cup v$
    Fusionner les sommets de $\mathcal{F}$
    $\mathcal{O} = \mathcal{O} \backslash \mathcal{F}$

Précisions $\mathcal{P}(k)$

$\rightarrow$ Classe de k : (α, β)

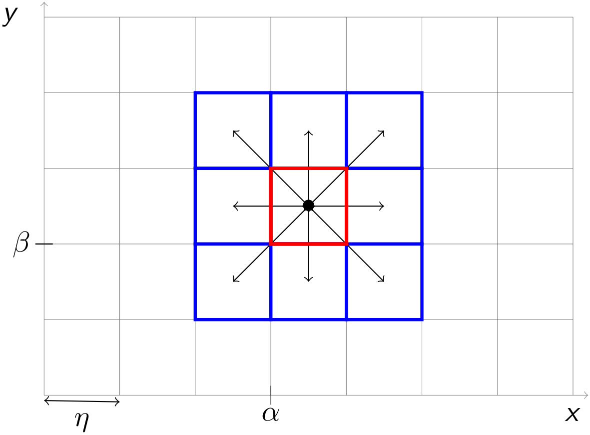
$\rightarrow$ $\mathcal{P}(k) :=$ Sommets des classes contiguës à celle de k, proches à ϵ près



Précisions $\mathcal{P}(k)$

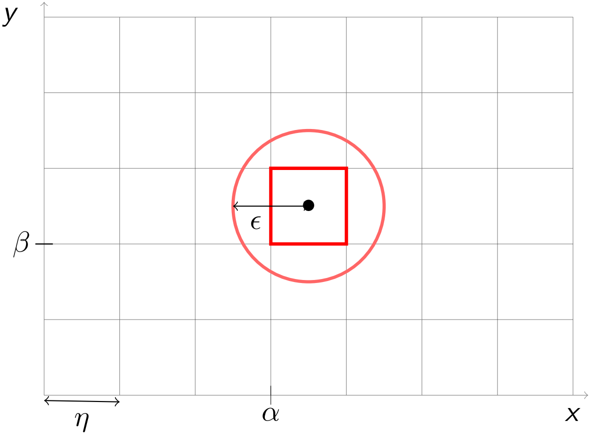
→ Classe de k : (α, β)

→ $\mathcal{P}(k) :=$ Sommets des classes contiguës à celle de k, proches à ϵ près



Précisions $\mathcal{P}(k)$

- → Classe de k : (α, β)
- → $\mathcal{P}(k) :=$ Sommets des classes contiguës à celle de k, proches à ϵ près

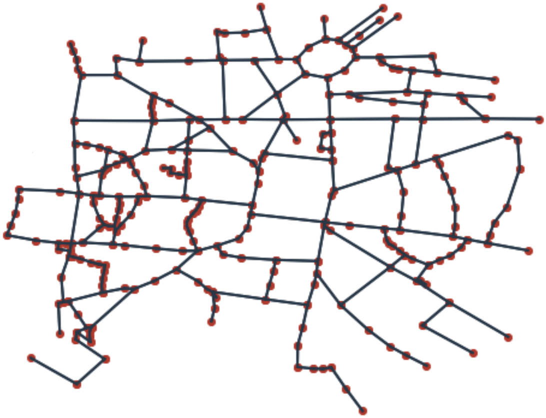


Une fois le pré-traitement achevé...

- $917 \rightarrow 485$

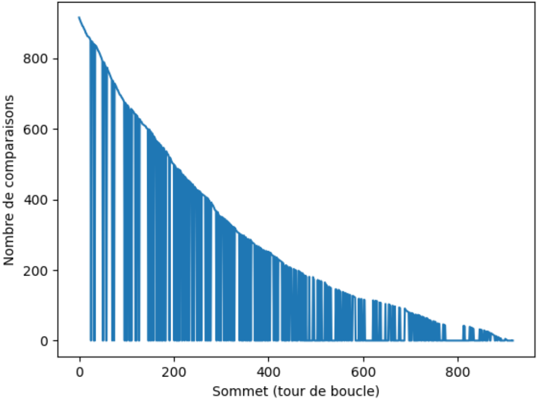
Une fois le pré-traitement achevé...

- $917 \rightarrow 485$



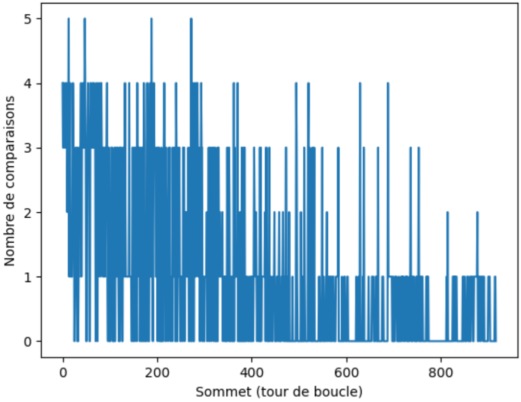
Première approche : complexité

- 200000 comparaisons



Seconde approche : complexité

- 930 comparaisons



Choix d'un modèle

On définit 3 types de rues :

→ Administrative : peu dense

→ Résidentielle : dense

→ Commerçante : très dense

Choix d'un modèle

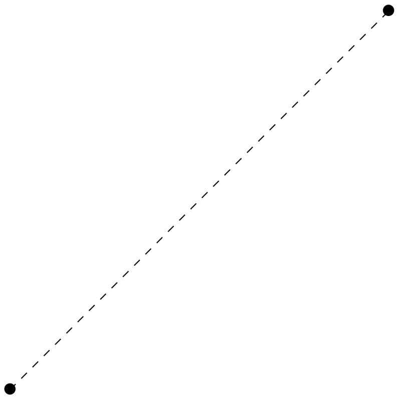
On définit 3 types de rues :

- → Administrative : peu dense
- → Résidentielle : dense
- → Commerçante : très dense

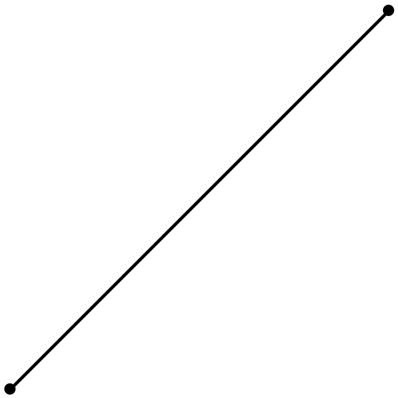
Adjonction de probabilités :

- → $\mathbb{P}(\mathcal{A}) = 0.25$
- → $\mathbb{P}(\mathcal{R}) = 0.25$
- → $\mathbb{P}(\mathcal{C}) = 0.5$

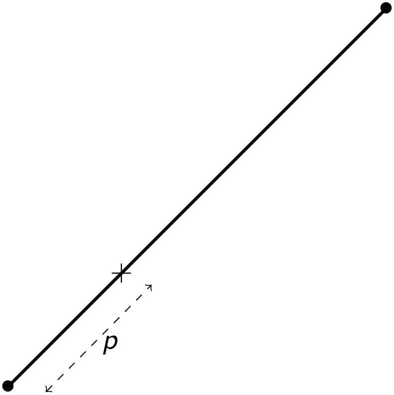
Processus



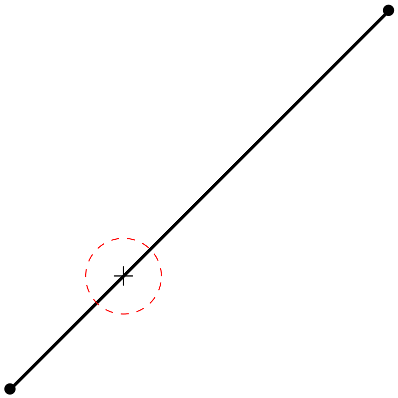
Processus



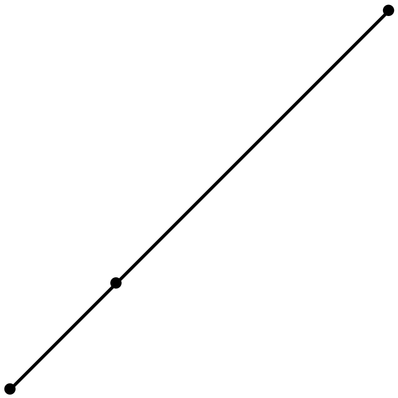
Processus



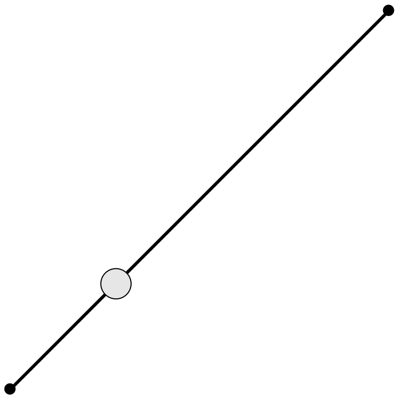
Processus



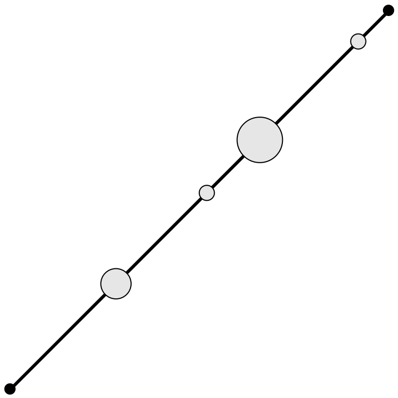
Processus



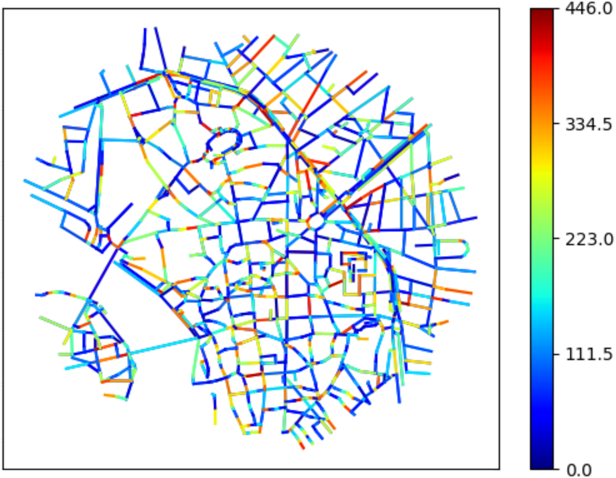
Processus



Processus

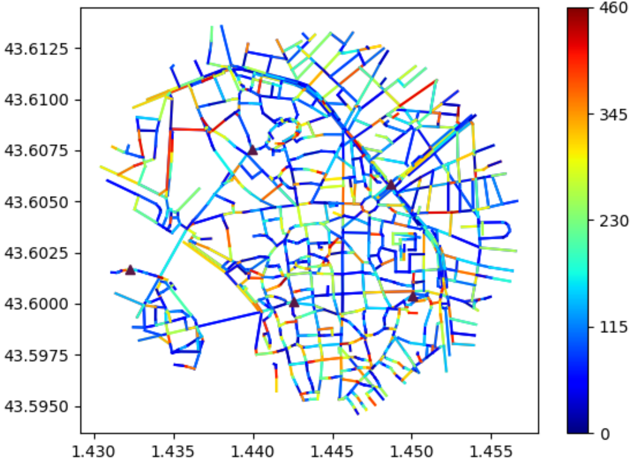


Graphe pondéré



Résolution

Application de la méthode du recuit simulé sur une instance valide



Conclusion

$\rightarrow$ Une problématique, une résolution

$\rightarrow$ Parachèvement : déterminer un parcours du réseau

```
import sqlite3 as sql
import matplotlib.pyplot as plt
import math as m
import numpy as np
import random
import json
from mpl_toolkits.axes_grid1 import make_axes_locatable
import matplotlib as mpl
import matplotlib.pyplot as plt
#--------------------------------------------------
#           Fonctions auxiliaires
#--------------------------------------------------
d_global=1000
def est_valide (tab):
    #Retourne si dist(tab,capitole)<p Km où tab=[long,lat]
    lat=m.radians(tab[1])
    long=m.radians(tab[0])
    r=6371*(10**3)
    lat_t=m.radians(43.60395967066511)
    long_t=m.radians(1.4433469299842416)

    ↪  d=2*r*m.asin(m.sqrt((m.sin((lat-lat_t)/2)**2)+m.cos(lat)*m.cos(lat_t)*(m.sin((long-long_t)/2
    return (d<d_global)

def distance (a,b):
    #Calcule d(a,b) en m
    lat=m.radians(a[1])
    long=m.radians(a[0])
    r=6371*(10**3)
    lat_t=m.radians(b[1])
    long_t=m.radians(b[0])

    ↪  d=2*r*m.asin(m.sqrt((m.sin((lat-lat_t)/2)**2)+m.cos(lat)*m.cos(lat_t)*(m.sin((long-long_t)/2
    return d
```

```python
#----------------------------------------------------
#   Fonctions d'affichage
#----------------------------------------------------

def affiche_graphe_simple( d : dict , coord : list , opt : bool) :
  #Affiche le graphe d avec les noeuds si opt = true
  for k,v in d.items():
     for kk in v.keys():
        y=[coord[k][0],coord[kk][0]]
        x=[coord[k][1],coord[kk][1]]
        plt.plot(x,y,c='#283747')
        if opt :
          plt.scatter(x,y,s=15,c='#B03A2E')
  plt.show()


def coord_stations(s:list,tab : list):
  (x,y,d)=s
  A=tab[x]
  B=tab[y]
  L=distance(tab[x],tab[y])
  alpha=d/L
  AB=[B[0]-A[0],B[1]-A[1]]
  C=[A[0]+alpha*AB[0],A[1]+alpha*AB[1]]
  return C
```

```python
def affiche_graphe_pondération( d : dict, tab : list,station : list ):
  #Affiche le graphe d et les stations
  # tab: coordonnées
    N = max_pop
    fig, ax1 = plt.subplots()
    # colormap
    cmap = plt.get_cmap('jet', N)
    c=0
    for k,v in d.items():
        for j in v.keys():

            x=[tab[j][0],tab[k][0]]
            y=[tab[j][1],tab[k][1]]
            ax1.plot(x, y, c=cmap(popL[c]))

            c+=1

    for i in range(0,len(station)):
      C=coord_stations(station[i],tab)
      ax1.plot(C[0],C[1],marker='^',c='#581845')



    # Normalizer
    norm = mpl.colors.Normalize(vmin=0, vmax=N)

    # creating ScalarMappable
    sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
    sm.set_array([])

    plt.colorbar(sm, ticks=np.linspace(0, N, 5))
    plt.show()
```

```
#--------------------------------------------------#
#           Récupération des données               #
#--------------------------------------------------#

#-Base de donnée support --------------------------
con = sql.connect('filaire_de_voirie.db')
requete2="""select geo_shape FROM "filaire-de-voirie" """


LGeopoints=[] # Tableau brut des données récupérées : géopoints


#--------------------------------------------------#
#           partitionnement des données            #
#--------------------------------------------------#
gd = {}

gd["longmin"]=1.4433469299842416
gd["longmax"]=1.4433469299842416
gd["latmin"]=43.60395967066511
gd["latmax"]=43.60395967066511

def taille_data (gd : dict, tab : list):
    if tab[0]<gd["longmin"]:
        gd["longmin"]=tab[0]
    if tab[0]>gd["longmax"]:
        gd["longmax"]=tab[0]
    if tab[1]<gd["latmin"]:
        gd["latmin"]=tab[1]
    if tab[1]>gd["latmax"]:
        gd["latmax"]=tab[1]
```

```
def partition_vide(g : list, l : int , L : int ):

    for i in range(0,l):
        g.append([])
    for i in range(0,l):
        for j in range(0,L):
            g[i].append([])
#-----------------------------------------------------
def recupere_data (f,con, req) :
    #Execute la requete req dans la db passé par con
    curseur = con.cursor()
    res = None
    curseur.execute(req)
    res = curseur.fetchall()

    for i in range(0,len(res)-1):
        a=eval(res[i][0])
        if est_valide(a['coordinates'][0][0]):
            f.append(a['coordinates'][0])
            for i in a['coordinates'][0]:
                taille_data(gd,i)
    return f


partnoeuds=[]

#On récupère les données dans f grâce à la requête requete2
t=recupere_data(LGeopoints,con,requete2)
```

```
#--------------------------------------------------------
#            Traitement des données
#--------------------------------------------------------



# 1/ Création du graphe

g={} #Graphe dict de dict
tab=[] #Coordonnées associées à un pt du graphe
noeud_valide=[] # Un noeud est dit valide (ie à "true") ssi il est à considérer après fusion
↪   du graphe

def ajoute_voisin(g : dict, i : int , c : tuple ) :
    #Ajoute la relation i->j et j->i telle que d(i,j)=dist
    (j,dist)=c
    if(not(j in g[i])):
        g[i][j]=dist
        g[j][i]=dist
    else:
        print(i,j)
        raise ValueError("Relation déjà présente")
```

```python
def ajoute_noeud(g : dict, t : list, tab : list):
    n=0 #nombre de noeuds

    for i in range(0,len(t)):

        for j in range(0,len(t[i])):
            g[n]={}
            tab.append(t[i][j])
            noeud_valide.append(True)
            n+=1
        #Complétion du voisinnage
        if (len(t[i]))>1:
            for j in range (0,len(t[i])-1):
                # On ajoute les relations ( j <-> j+1 ) entre les sommets intermédiaire d'une
                ↪  même route
                ajoute_voisin(g,n-len(t[i])+j,(n-len(t[i])+j+1,distance(t[i][j],t[i][j+1])))


#On complete le graphe g grâce aux données récupérées dans t
#en retenant les coordonnées de chaque noeud de g dans tab
ajoute_noeud(g,t,tab)
```

```python
# 2/ Complétion de la partnoeuds

##Calcul distance partition
eta = 3 #distance de partionnement
nlat=1+int(distance([gd["longmin"],gd["latmin"]],[gd["longmin"],gd["latmax"]])/eta)
nlong=1+int(distance([gd["longmin"],gd["latmin"]],[gd["longmax"],gd["latmin"]])/eta)

partition_vide(partnoeuds,nlong,nlat) # On crée la partition vide

tabclasse=[] # Pour chaque noeud, on retient sa classe.

def add_part(gg : dict, t: list):
    # Etant donné un graphe "gg" , on partionnent l'ensemble des noeuds selon leurs coordonées
    ↪  dans "t"
    for k in gg.keys():
        N=0
        E=0
        dlong=distance([gd["longmin"],tab[k][1]],tab[k])
        dlat=distance([tab[k][0],gd["latmin"]],tab[k])
        while(not(N*eta<=dlong and dlong<=(N+1)*eta)):
            N+=1
        while(not(E*eta<=dlat and dlat<=(E+1)*eta)):
            E+=1
        partnoeuds[N][E].append(k) #On ajoute le noeud dans la partition
        tabclasse.append([N,E]) #On retient la classe du noeud
    print("Création de la partition terminée")

add_part(g,tab)
```

```
# 3/ Correction du graphe


def fusion ( i : int , j : int ):
    # Fusionne le noeud i et j, en attibuant remplaçant j par i dans les voisins de celui-ci
    for k,v in g[j].items():
        if (k!=i and not(k in g[i])):
            g[i][k]=v
        g[k][i]=v
        del g[k][j]

    noeud_valide[j]=False

def apptab(x : int, y : int ):
    # part[x][y] isn't out of range
    return(x>=0 and y>=0 and x<nlong and y<nlat)
```

```
def classement(g:dict, t : list , eps : float):
    ## Classe le dictionnaire g en recquérant les fusions proches à eps près
    ## Précondition : eps<eta

    for k in g.keys():

        if noeud_valide[k]: #Teste si k n'a pas déjà été fusionné

            X=k
            a_fusionner=[]
            # a_fusionner U {X} est l'ensemble des noeuds à distance < esp déjà trouvés
            # X est celui d'identifiant minimal
            n,e=tabclasse[k]

            for i in range(-1,2):
                for j in range(-1,2):
                    if apptab(n+i,e+j):
                        for z in partnoeuds[n+i][e+j]:
                            if noeud_valide[z] and distance(t[z],t[k])<eps and z!=X:
                                if z<X:
                                    a_fusionner.append(X)
                                    X=z
                                else:
                                    a_fusionner.append(z)
            for i in a_fusionner:
                fusion(X,i)
                assert(not(noeud_valide[i]))
```

```
classement(g,tab,0.9*eta) # Attention eps < eta

g_final={}
gf_coord=[]

def renum( t1 : list , t2 : list, g1 : dict, gf : dict):
    table=[]
    c=0
    for k in g1.keys():
        if t1[k]:
            table.append(c)
            t2.append(tab[k])
            c+=1
        else:
            table.append(-1)

    for k,v in g1.items():
        if table[k]!=-1:
            o=int(table[k])
            gf[o]={}
            for kk,vv in v.items():
                if table[kk]!=-1:
                    p=int(table[kk])
                    gf[o][p]=vv



renum(noeud_valide,gf_coord,g,g_final)

#print("Graphe final crée avec succès !")
```

```python
#4/ Traitement du graphe

## --------- Ajout de la population ------------ #

immeubleL = [] ## Liste d'immeubles
poidsL = [] # Poids de chaque immeuble (par rue !!!! [[rue1],[rue2]] )
popL = [] # Nombre d'habitant par rue
poidsLL = [] #Poids de chaque immeuble
## Types de rues :
# 0 : Administrative
# 1 : Résidentielle
# 2 : Commerçante

def proba_rue() :
# Retourne un type de rue
    x=random.random()
    if x<0.5:
        y=random.random()
        if y<0.5:
            return 0
        else: return 2
    else: return 1
```

```
##dictinnaire d'arete traitée
def ajoute_population( g : dict , poids_immeuble : list , immeuble : list, population : list)
↪  :
## g est une copie du graphe
    for k,v in g.items() :
        for kk,vv in v.items():
            print("Traitement de la rue :",k,"->",kk)
            if vv>=0:
                # On définit le type de la rue
                t=proba_rue()
                poids_im=[]
                dist_parcours=0
                pop_rue=0
                while(dist_parcours<vv):

                    poids=0
                    # Un immeuble tous les px +/- pt
                    pt=0
                    px=0
                    match t:
                        case 0:
                            px=vv/5
                            pt=random.uniform(0,0.5*px)
                            px+=pt
                            poids=random.randint(20,30)
                        case 1:
                            px=vv/10
                            pt=random.uniform(0,0.5*px)
                            px+=pt
                            poids=random.randint(40,60)
                        case 2:
                            px=vv/7
                            pt=random.uniform(0,0.8*px)
                            px+=pt
                            poids=random.randint(20,40)
```

```
                    if dist_parcours+px<vv:
                        immeuble.append((k,kk,dist_parcours+px))
                        poids_im.append(poids)
                        poidsLL.append(poids)
                        pop_rue+=poids
                        dist_parcours=dist_parcours+px

                    else:
                        break
                    poids_immeuble.append(poids_im)
                    population.append(pop_rue)

ajoute_population(g_final,poidsL,immeubleL,popL)

max_pop=0

for i in range(0,len(popL)):
    if max_pop <= popL[i]:
      max_pop=popL[i]
```

```
# -------------------------------------#
#               Résolution             #
#--------------------------------------#

import heapq

def dijkstra (g,s) :
  n = len(g)
  d = []
  for i in range (0,n) :
    d.append(10000)
  d[s]=0
  o=[]
  heapq.heappush(o, (d[s], s))
  while o != [] :
    (_,u) = heapq.heappop(o)
    for k in g[u].keys() :
      if d[u]+(g[u])[k] < d[k] :
        temp = d[k]
        d[k] = d[u]+(g[u])[k]
        if temp == 10000 :
          heapq.heappush(o,(d[k],k))
  return d

distances_gf = []
for i in range (0,len(g_final)) :
    distances_gf.append(dijkstra(g_final,i))
```

```
def dist_imm_sta (imm,sta,g) :
    (s1_imm,s2_imm,d_imm) = imm
    (s1_sta,s2_sta,d_sta) = sta
    if (s1_imm==s1_sta) and (s2_imm==s2_sta) :
        return max(d_imm,d_sta)-min(d_imm,d_sta)
    else :
        return
        ↪ min(distances_gf[s1_imm][s1_sta]+d_imm+d_sta,distances_gf[s1_imm][s2_sta]+d_imm+(g[s1_st



def min_dist (g,l,l3,p) :
  res = dist_imm_sta(l3[p],l[0],g)
  for i in range (1,len (l)) :
    res = min(res,dist_imm_sta(l3[p],l[i],g))
  return res

def f (g,l,l2,l3) :
  res = 0
  som = 0
  for i in range (0,len(l3)) :
    res = res + (min_dist(g,l,l3,i)*l2[i])
  for i in range (0,len(l3)) :
    som = som + l2[i]
  res = res/som
  return res
```

```python
def simulated_annealing(g,initial_state,l2,l3):
    """Peforms simulated annealing to find a solution"""
    initial_temp = 300
    final_temp = 1
    current_temp = initial_temp

    # Start by initializing the current state with the initial state
    current_state = initial_state
    solution = current_state
    nb_sta = len(initial_state)
    while current_temp > final_temp:
        neighbor = get_neighbors(g,current_state,l2,(current_temp*d_global)/(nb_sta*300))
        # print(neighbor)
        # Check if neighbor is best so far
        cost_diff = f(g,current_state,l2,l3)-f(g,neighbor,l2,l3)
        # print(cost_diff)
        # if the new solution is better, accept it
        if cost_diff > 0:
            solution = neighbor
        # if the new solution is not better, accept it with a probability of e^(-cost/temp)
        else:
            if random.uniform(0, 1) > m.exp(cost_diff / current_temp) :
                solution = neighbor
        # decrement the temperature
        # print(solution)
        current_temp = current_temp * 0.99
        current_state = solution
    return solution
```

```python
def get_neighbor_station(g,station,eps):
  """Returns neighbors of the argument state for your solution."""
  (s1,s2,d) = station
  carburant=random.uniform(0,eps)
  sens=random.randint(0,1)
  if sens == 1 :
    d = (g[s1])[s2] - d
    temp = s1
    s1 = s2
    s2 = temp
  while carburant != 0 :
    if carburant > (g[s1])[s2] - d :
      carburant = carburant - ((g[s1])[s2] - d)
      voisins = list(g[s2].keys())
      temp=s2
      s2=random.choice(voisins)
      s1=temp
      d = 0
    else :
      d = d + carburant
      carburant = 0
  return (s1,s2,d)


def get_neighbors(g,state,l2,eps):
  """Returns neighbors of the argument state for your solution."""
  res=[]
  for i in range (0,len(state)):
    res.append(get_neighbor_station(g,state[i],eps))

  return res
```

```
def best_answer (g,initial_state,l2,l3) :

  res = simulated_annealing(g,initial_state,l2,l3)

  print(res)
  for i in range (0,4) :
    new_res = simulated_annealing(g,initial_state,l2,l3)
    print(new_res)
    if f(g,new_res,l2,l3) < f(g,res,l2,l3) :
      res = new_res
  return res

tailleim=len(immeubleL)
si=[]
for i in range (0,5):
   o=random.randint(0,tailleim)
   si.append(immeubleL[o])
```

```python
# ---------------------------------------#
#        Lancement des processus         #
#----------------------------------------#
answer = best_answer(g_final,si,poidsLL,immeubleL)
affiche_graphe_pondération(g_final,gf_coord,answer)
#print(si)
print("init:",f(g_final,si,poidsLL,immeubleL))
#print(answer)
print("res",f(g_final,answer,poidsLL,immeubleL))
```