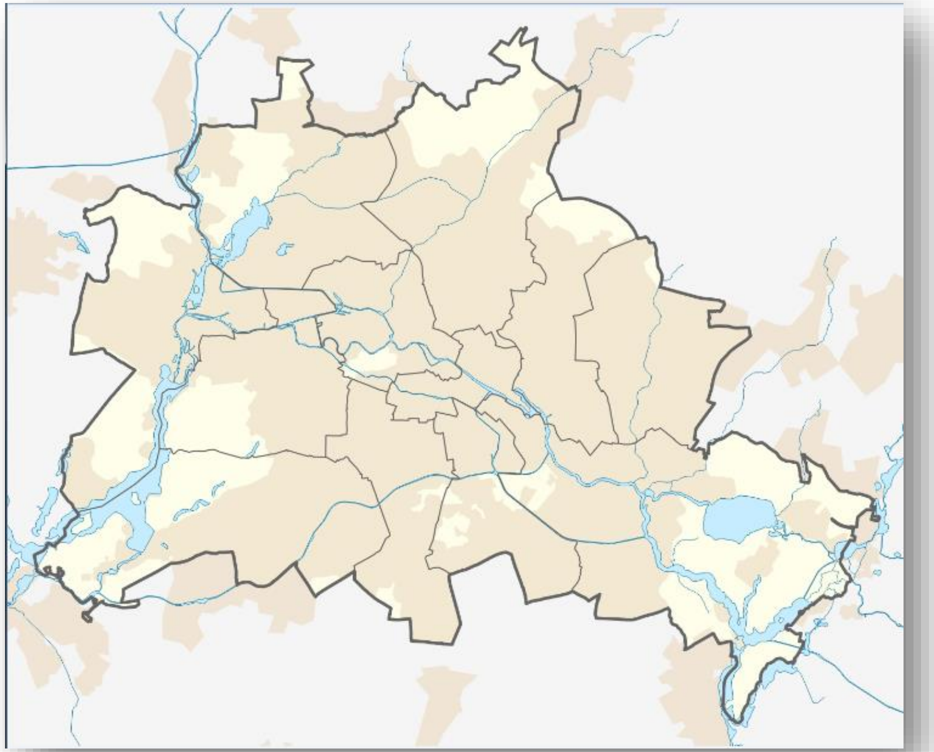
Théorie des Graphes
TP Dijkstra – A✫

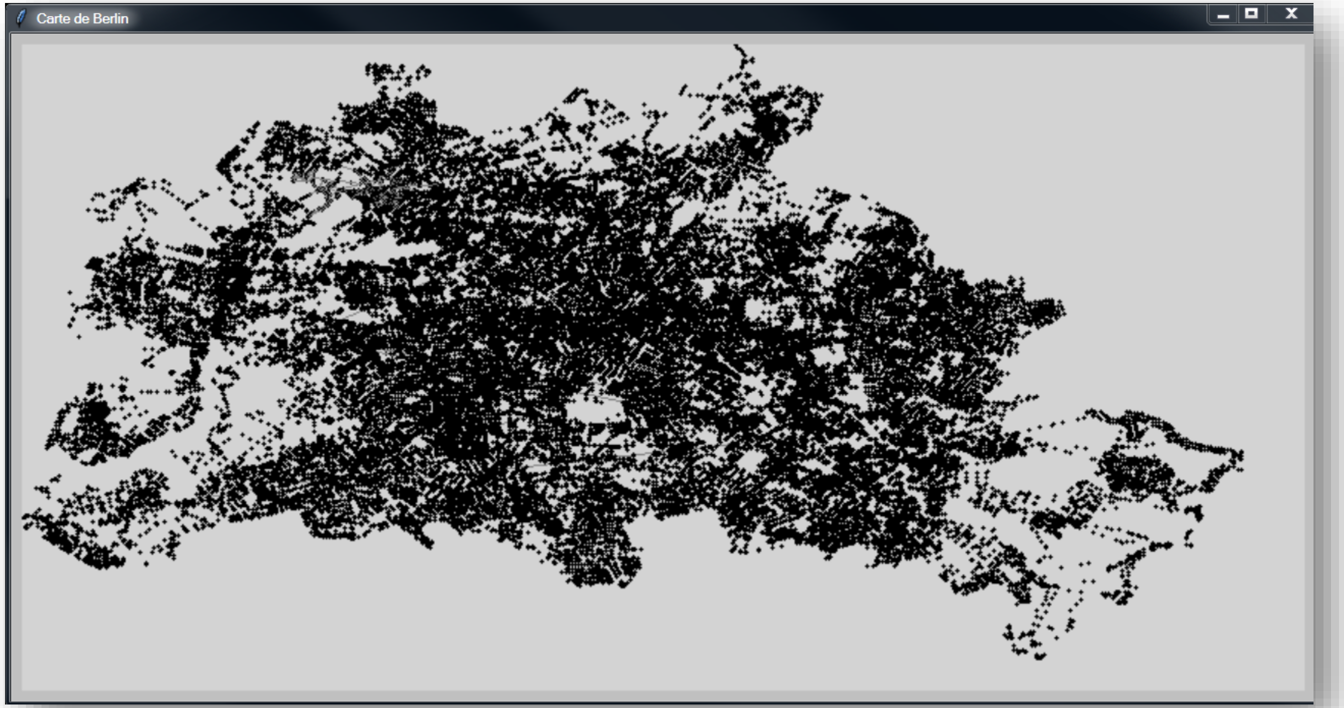
Polytech Tours

Environnement

Environnement

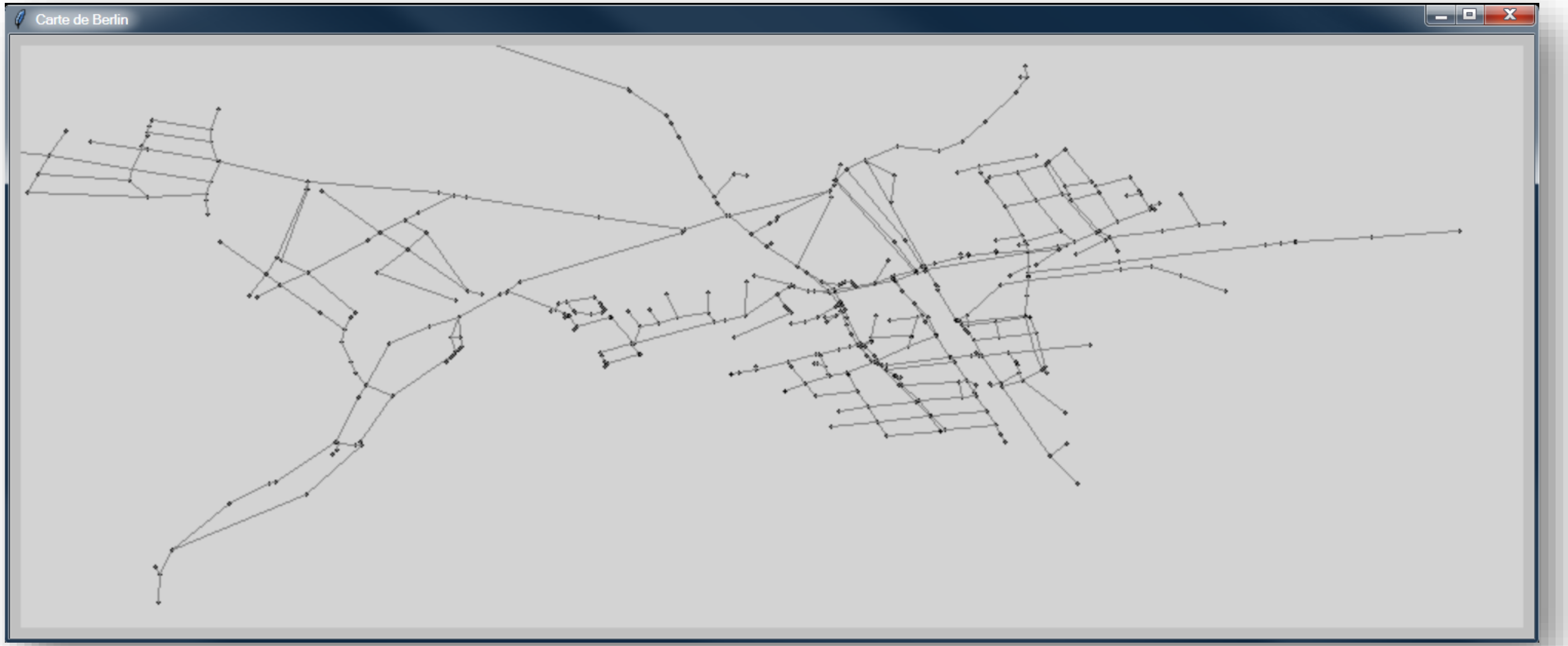
- Tous les TP se feront sur un graphe unique.
- Ce graphe représente le plan de la ville de Berlin





Environnement

- Le graphe comporte 59673 nœuds (`berlin_noeuds.txt`) et 145840 arcs (`berlin_arcs.txt`).
- On travaillera d'abord sur un graphe partiel (« toy ») comportant seulement 444 sommets et 977 arcs, avant de tester ses algorithmes sur le graphe complet.



Environnement

Fichiers

- Le booléen « `graphe_toy` » s'il est à 1 fera ouvrir le fichier d'exercice, s'il est à 0 fera ouvrir le graphe complet.

```python
# ###########################################################
# Lecture des fichiers
# ###########################################################

graphe_toy = 1



if graphe_toy:
    fichier_noeuds = 'toy_noeuds.txt' # 'berlin_noeuds.txt'
    fichier_arcs = 'toy_arcs.txt' # 'berlin_arcs.txt'
    zoom = 10
else:
    fichier_noeuds = 'berlin_noeuds.txt' # 'toy_noeuds.txt' # 'berlin_noeuds.txt
    fichier_arcs = 'berlin_arcs.txt' # 'toy_arcs.txt' # 'berlin_arcs.txt'
    zoom = 1
```

Avec Networkx

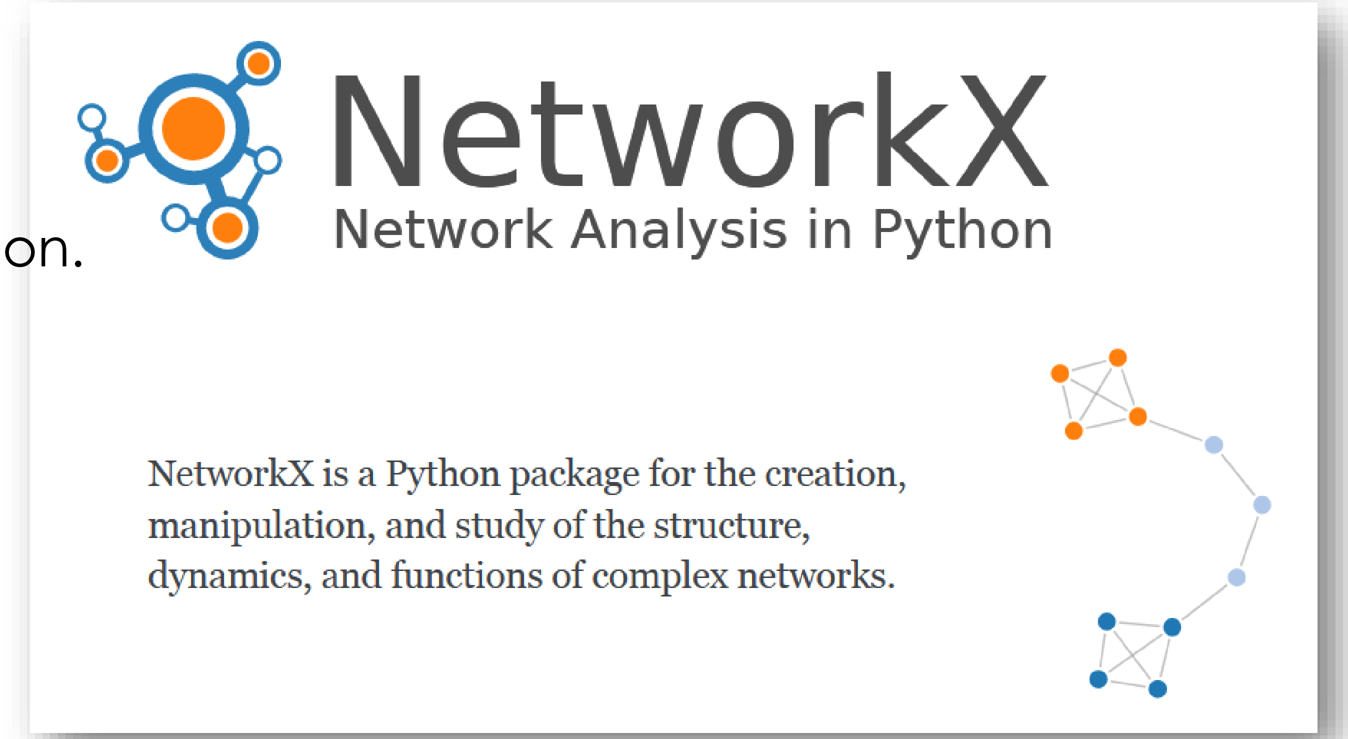
Environnement

Package

- On utilisera le package `NetworkX` de Python.

```
import networkx as nx
```

- On fera 2 choses pour chaque TP :
  1. On codera l'algorithme
  2. On appellera la méthode du package
  3. On comparera l'efficacité de notre code avec celui de NetworkX



- Selon le problème étudié, on utilisera la classe `Graph()` ou la classe `DiGraph()` (« directed graph », graphe orienté). Attention, selon le cas, les méthodes que l'on peut appeler ne sont pas les mêmes.

```
from tkinter import *
from math import *
import os
from numpy import *
import tkinter.font as tkFont
import time
import networkx as nx
```

Environnement

Tutorial NetworkX – Getting started

- https://networkx.guide/getting-started/
- https://networkx.guide/functions
- Etc.

Environnement

Lire les nœuds du graphe

- Ouvrir le fichier `fichier_noeuds,` lire toutes les lignes avec `readlines()`, fermer le fichier

```
fn = open(fichier_noeuds,"r")
ln = fn.readlines()
fn.close()
```

- Créer le graphe
- Ajouter 2 attributs pour les coordonnées

```
G = nx.Graph()
nx.set_node_attributes(G,0,"X")
nx.set_node_attributes(G,0,"Y")
```

- Définir les sommets du graphe

```
idx=0
for l in ln:
    ll = l .split("\t")
    G.add_node(idx, X=float(ll[1]),Y=float(ll[2]))
    idx+=1
```

toy_noeuds.txt - Bloc-notes

Fichier Edition Format Affichage ?

0	225.6501182033121	98.93617021276674
1	226.9503546099296	98.69976359338648
2	224.94089834515455	99.0543735224606
3	227.06855791962158	96.45390070922559
4	229.07801418439854	97.28132387706762
5	224.5862884160786	98.46335697399877
6	219.73995271867702	100.47281323877195
7	229.55082742316836	98.10874704491715
8	229.90543735224807	96.80851063829965
9	224.5862884160786	97.16312056738127
10	222.57683215129975	98.81796690307283
11	219.73995271867702	100.47281323877195
12	220.21276595744874	99.40898345154221
13	229.78723404255422	98.22695035461857
14	230.02364066194008	96.6903073286133
15	221.86761229314595	97.28132387706762
16	221.39479905437423	99.17257683215445
17	218.32151300236566	98.81796690307283
18	219.62174940898504	100.47281323877195
19	219.38534278959918	98.34515366430492
20	230.02364066194008	98.46335697399877
21	227.5413711583933	96.09929078014397
22	230.14184397163206	96.80851063829965
23	220.80378250591056	98.22695035461857
24	218.20330969267366	98.81796690307283
25	217.61229314420999	101.06382978723377
26	220.09456264775673	101.18203309692763
27	226.24113475177393	100.11820330969033
28	231.0874704491736	97.04491725767991
29	216.07565011820475	99.40898345154221
30	217.73049645390196	101.30023640661398
31	220.09456264775673	101.18203309692763
32	231.44208037825143	97.04491725767991
33	215.48463356974293	98.69976359338648

Environnement

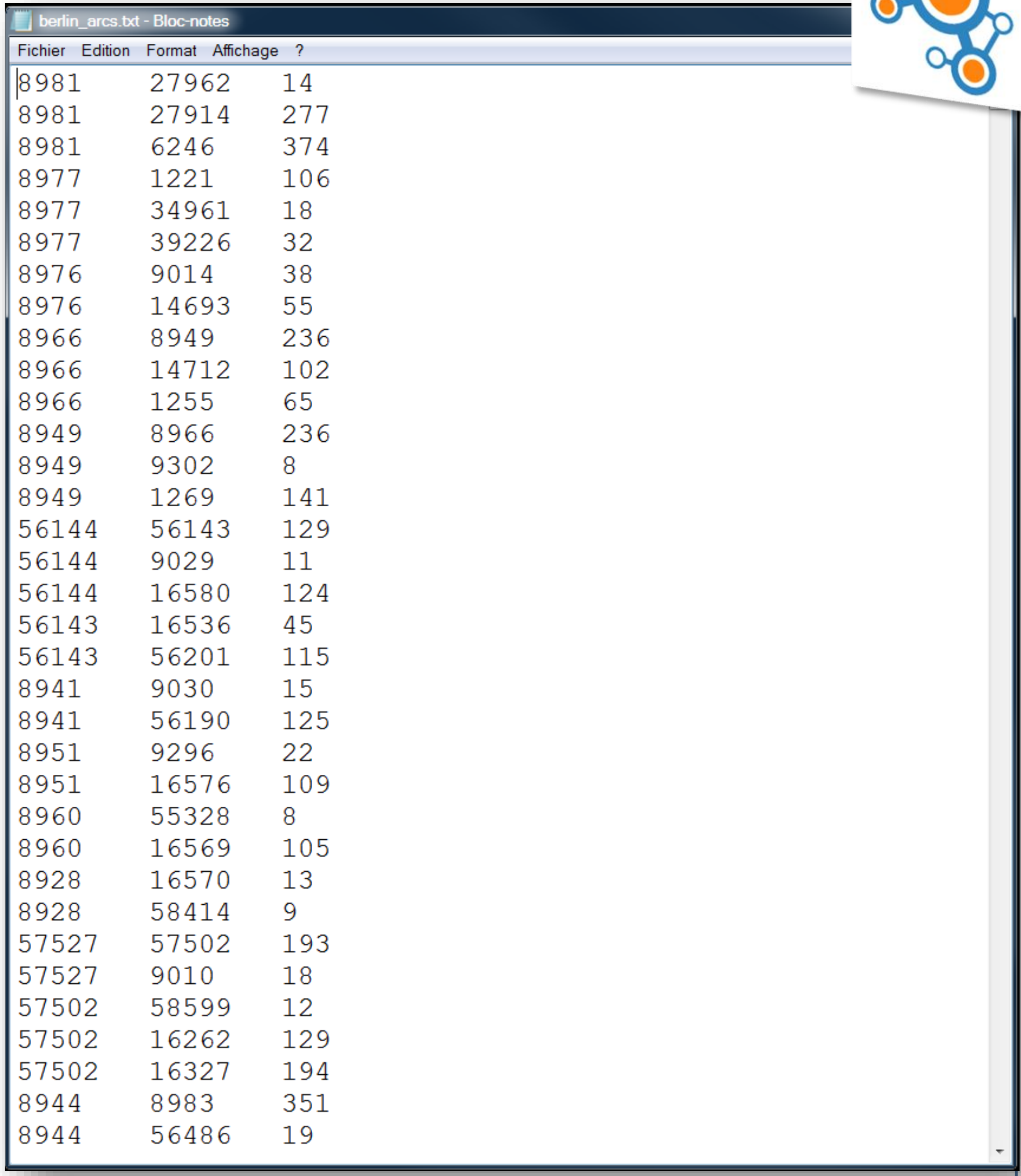
Lire les arcs du graphe

- Ouvrir le fichier `fichier_arcs`, lire toutes les lignes avec `readlines()`, fermer le fichier

- Définir les arcs du graphe

```
for l in ln:
    ll = l .split("\t")
    o = int(ll[0])
    d = int(ll[1])
    G.add_edge(o,d,weight=int(ll[2]))
```

- On appellera `NbNoeuds` le nombre de nœuds et `NbArcs` le nombre d'arcs (obtenus par `G.number_of_nodes()` et `G.number_of_edges()`)



8981	27962	14
8981	27914	277
8981	6246	374
8977	1221	106
8977	34961	18
8977	39226	32
8976	9014	38
8976	14693	55
8966	8949	236
8966	14712	102
8966	1255	65
8949	8966	236
8949	9302	8
8949	1269	141
56144	56143	129
56144	9029	11
56144	16580	124
56143	16536	45
56143	56201	115
8941	9030	15
8941	56190	125
8951	9296	22
8951	16576	109
8960	55328	8
8960	16569	105
8928	16570	13
8928	58414	9
57527	57502	193
57527	9010	18
57502	58599	12
57502	16262	129
57502	16327	194
8944	8983	351
8944	56486	19

Environnement

Dessiner le graphe

- Saisir les fonctions ci-contre et la définition du canvas

- Terminer votre code par :

```
can.pack() #Affiche le Canvas
fen.mainloop()
```

- On obtient les cartes souhaitées.

```
# ##########################################################
# Dessin du graphe
# ##########################################################
print('*** Dessin du graphe ***')

def TraceCercle(j,couleur,rayon,ep):
    x0 = (G.nodes[j]["X"]-minX)*zoom
    y0 = (G.nodes[j]["Y"]-minY)*zoom
    can.create_oval(x0-rayon, y0-rayon, x0+rayon, y0+rayon, \
                    outline = couleur, fill = couleur, width=ep)

def TraceArc(j1,j2,couleur,ep):
    x1 = (G.nodes[j1]["X"]-minX)*zoom
    y1 = (G.nodes[j1]["Y"]-minY)*zoom
    x2 = (G.nodes[j2]["X"]-minX)*zoom
    y2 = (G.nodes[j2]["Y"]-minY)*zoom
    can.create_line(x1,y1,x2,y2,fill = couleur,width=ep)

fen = Tk()
fen.title('Carte de Berlin')
coul_fond = "white"
#['purple','cyan','maroon','green','red','blue','orange','yellow']
coul_noeud = "black"

border = 20         # taille en px des bords

infini = 999999
maxX = maxY = 0
minX = minY = infini
for j in range(NbNoeuds):
    maxX = max(maxX,G.nodes[j]["X"])
    minX = min(minX,G.nodes[j]["X"])
    maxY = max(maxY,G.nodes[j]["Y"])
    minY = min(minY,G.nodes[j]["Y"])

Delta_X = (maxX-minX)*zoom
Delta_Y = (maxY-minY)*zoom
winWidth = Delta_X+2*border
winHeight = winWidth * Delta_Y / Delta_X

can = Canvas(fen, width = winWidth, height = winHeight, bg =coul_fond)

#  Affichage des noeuds et des arcs
rayon_noeud = 1                 # rayon pour dessin des points
for i in range(NbNoeuds):
    TraceCercle(i,coul_noeud,rayon_noeud,1)
    for s in G.neighbors(i):
        TraceArc(i,s,'grey',1)
```

« A la main »

MyCode

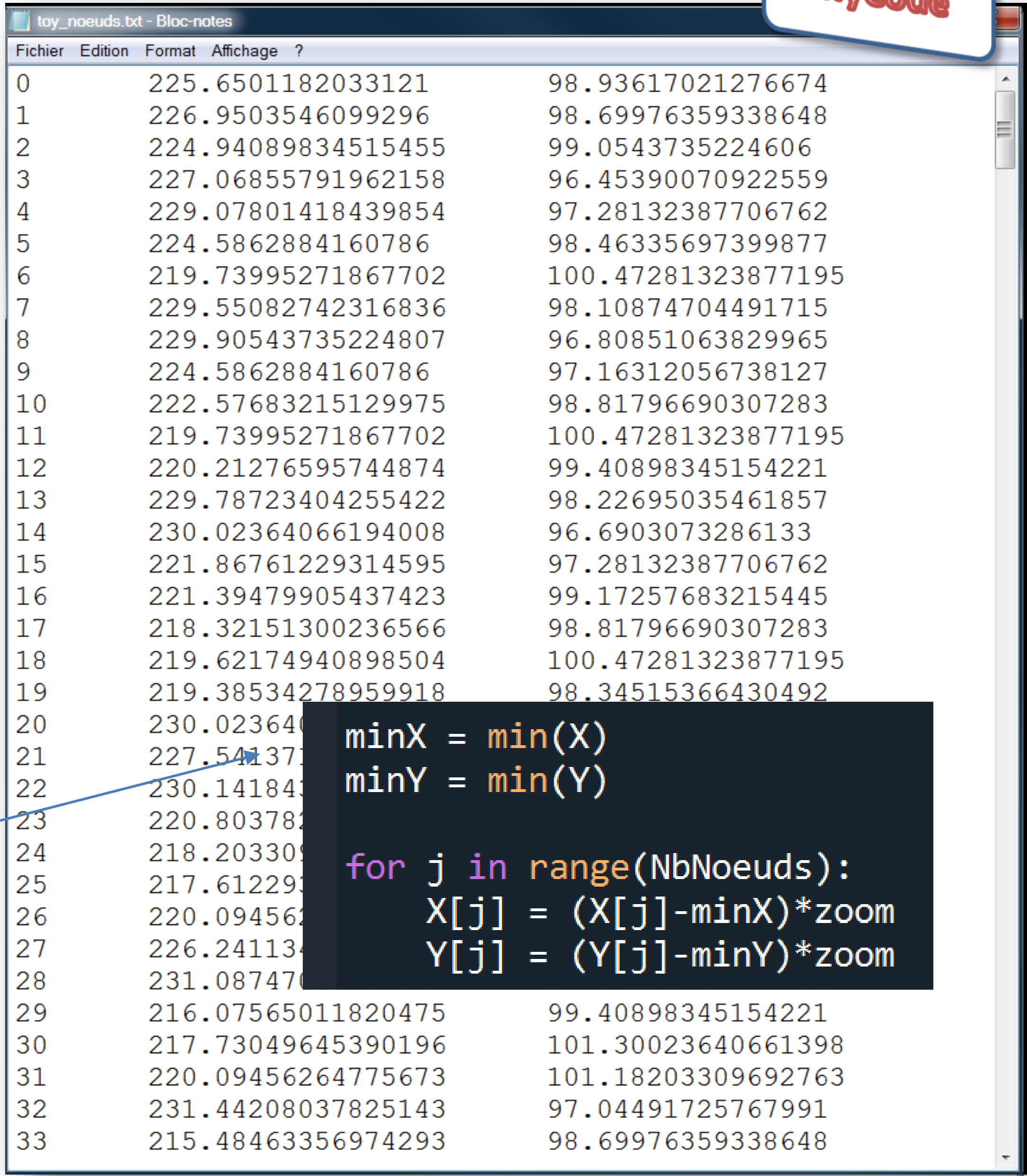
Environnement

Lire les nœuds du graphe

- Ouvrir le fichier `fichier_noeuds`, lire toutes les lignes avec `readline`**s**(), fermer le fichier

```
fn = open(fichier_noeuds,"r")
ln = fn.readlines()
fn.close()
```

- Créer 2 listes
  - `X = []`
  - `Y = []`
- Remplir ces listes (convertir les strings en float)
- Modifier `X` et `Y` pour l'affichage
- On appellera `NbNoeuds` le nombre de nœuds (taille de la liste `X`)



toy_noeuds.txt - Bloc-notes

Fichier Edition Format Affichage ?

0	225.6501182033121	98.93617021276674
1	226.9503546099296	98.69976359338648
2	224.94089834515455	99.0543735224606
3	227.06855791962158	96.45390070922559
4	229.07801418439854	97.28132387706762
5	224.5862884160786	98.46335697399877
6	219.73995271867702	100.47281323877195
7	229.55082742316836	98.10874704491715
8	229.90543735224807	96.80851063829965
9	224.5862884160786	97.16312056738127
10	222.57683215129975	98.81796690307283
11	219.73995271867702	100.47281323877195
12	220.21276595744874	99.40898345154221
13	229.78723404255422	98.22695035461857
14	230.02364066194008	96.6903073286133
15	221.86761229314595	97.28132387706762
16	221.39479905437423	99.17257683215445
17	218.32151300236566	98.81796690307283
18	219.62174940898504	100.47281323877195
19	219.38534278959918	98.34515366430492
20	230.02364	
21	227.54137	
22	230.14184	
23	220.80378	
24	218.20330	
25	217.61229	
26	220.09456	
27	226.24113	
28	231.08747	
29	216.07565011820475	99.40898345154221
30	217.73049645390196	101.30023640661398
31	220.09456264775673	101.18203309692763
32	231.44208037825143	97.04491725767991
33	215.48463356974293	98.69976359338648


```
minX = min(X)
minY = min(Y)

for j in range(NbNoeuds):
    X[j] = (X[j]-minX)*zoom
    Y[j] = (Y[j]-minY)*zoom
```

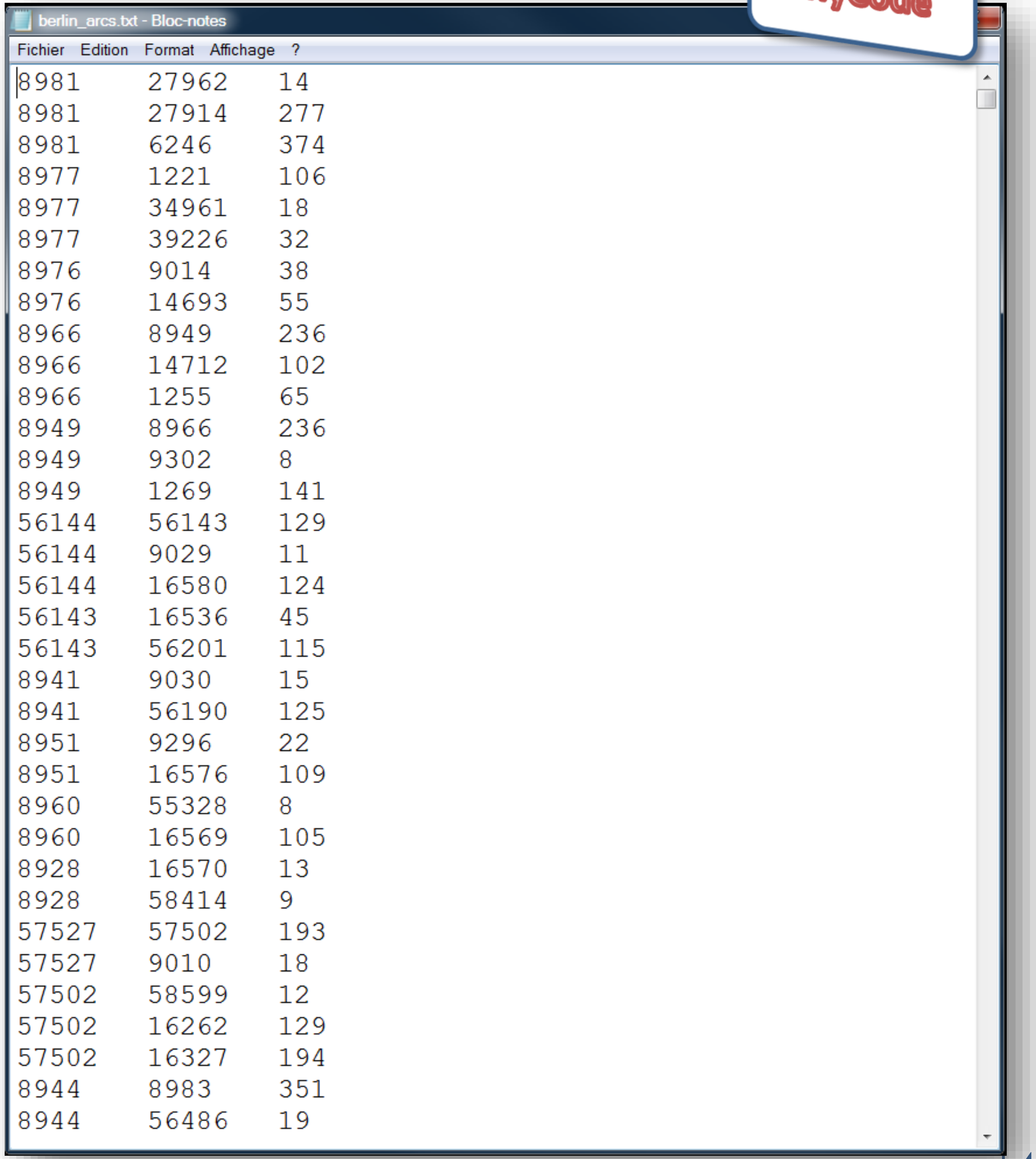
Environnement

Lire les arcs du graphe

- Ouvrir le fichier `fichier_arcs`, lire toutes les lignes avec `readlines()`, fermer le fichier

- Créer 3 listes :
  - `Origine = []`
  - `Destination = []`
  - `Longueur = []`
- Remplir ces listes (convertir les strings en int)

- On appellera `NbArcs` le nombre d'arcs


berlin_arcs.txt - Bloc-notes

Fichier Edition Format Affichage ?

8981	27962	14
8981	27914	277
8981	6246	374
8977	1221	106
8977	34961	18
8977	39226	32
8976	9014	38
8976	14693	55
8966	8949	236
8966	14712	102
8966	1255	65
8949	8966	236
8949	9302	8
8949	1269	141
56144	56143	129
56144	9029	11
56144	16580	124
56143	16536	45
56143	56201	115
8941	9030	15
8941	56190	125
8951	9296	22
8951	16576	109
8960	55328	8
8960	16569	105
8928	16570	13
8928	58414	9
57527	57502	193
57527	9010	18
57502	58599	12
57502	16262	129
57502	16327	194
8944	8983	351
8944	56486	19

Environnement

Dessiner le graphe

- Saisir les fonctions ci-contre et la définition du canvas

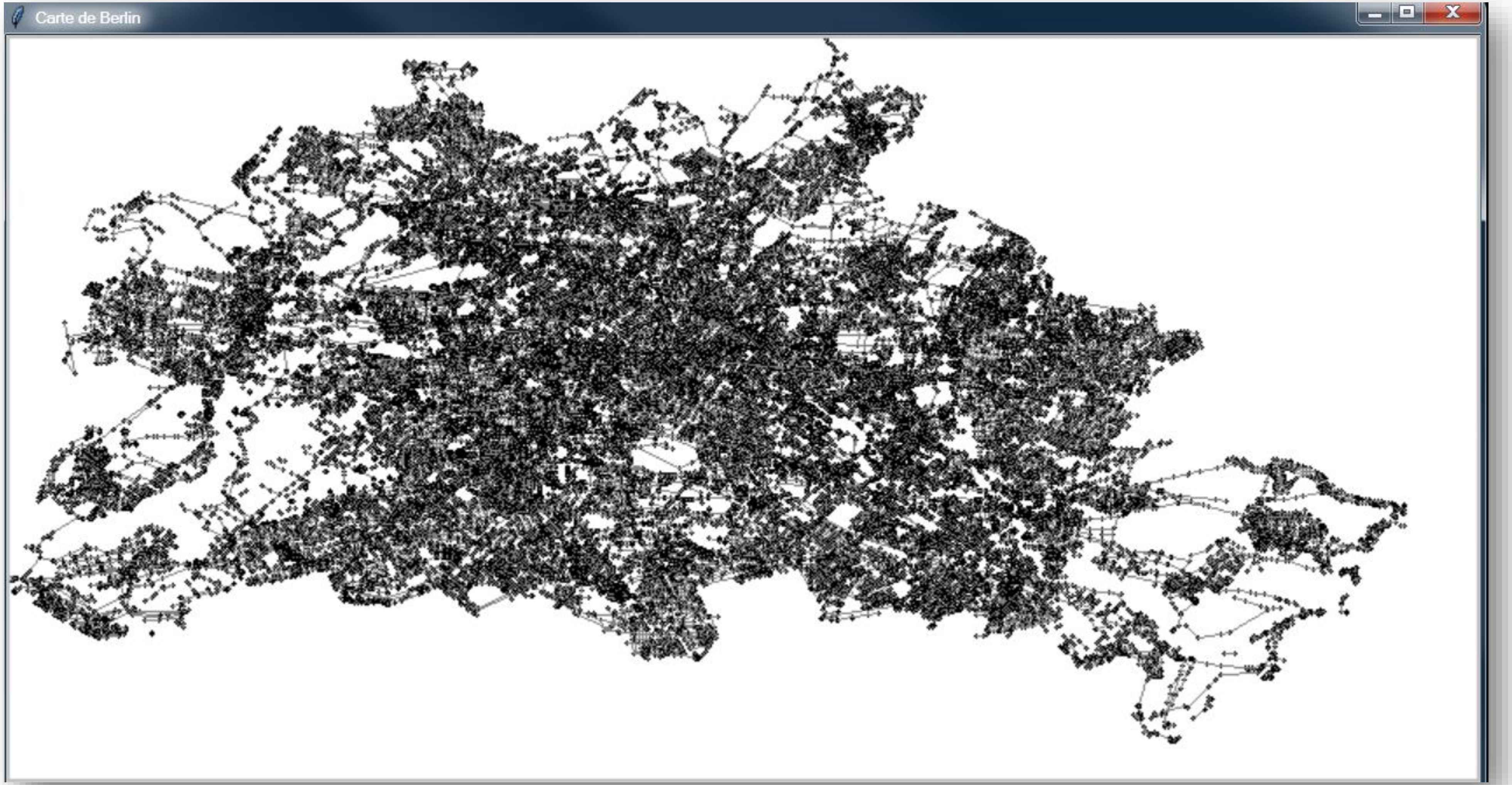
- Terminer votre code par :

```
can.pack() #Affiche le Can
fen.mainloop()
```

- On obtient les cartes souhaitées.

```
# ##########################################################
# Dessin du graphe
# ##########################################################
print('*** Dessin du graphe ***')

def TraceCercle(j,couleur,rayon,ep):
    can.create_oval(X[j]-rayon, Y[j]-rayon, X[j]+rayon, Y[j]+rayon, \
                    outline = couleur, fill = couleur, width=ep)

def TraceArc(j1,j2,couleur,ep):
    can.create_line(X[j1],Y[j1],X[j2],Y[j2],fill = couleur,width=ep)


fen = Tk()
fen.title('Carte de Berlin')
coul_fond = "lightgrey"
#['purple','cyan','maroon','green','red','blue','orange','yellow']
coul_noeud = "black"

border = 20         # taille en px des bords

Delta_X = max(X)-min(X)
Delta_Y = max(Y)-min(Y)
winWidth = Delta_X+2*border
winHeight = winWidth * Delta_Y / Delta_X

can = Canvas(fen, width = winWidth, height = winHeight, bg =coul_fond)

#  Affichage des noeuds et des arcs
rayon_noeud = 1               # rayon pour dessin des points
for i in range(NbNoeuds):
    TraceCercle(i,coul_noeud,rayon_noeud,1)
    for j in Succ[i]: TraceArc(i,j,'grey',1)
```

Environnement



Environnement

Lecture des fichiers
Création du graphe
Création du canvas
Affichage du graphe

inchangé

Code de chaque TP

Affichages du TP

- Soit faire 1 fichier .py par programme (avec et sans Networkx)
- Soit mettre les deux modes dans le même.

```
can.pack()
fen.mainloop()
```

inchangé

TP Dijkstra

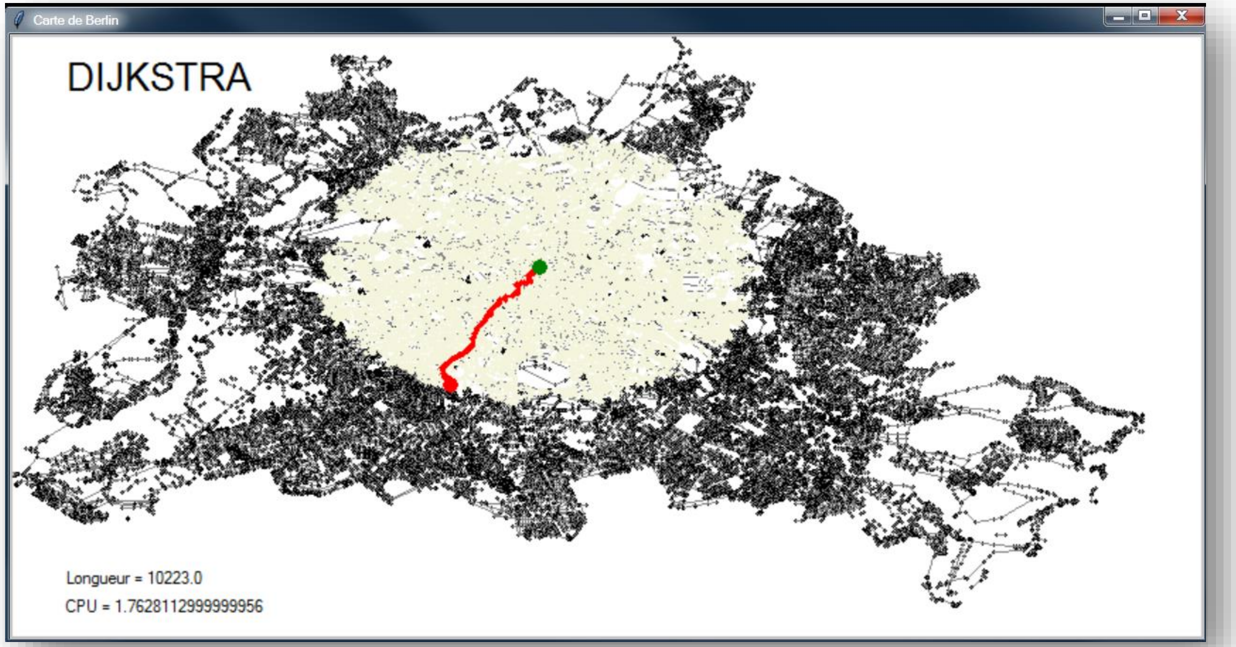
TP Dijkstra

Plus court chemin

1. Le but est de coder Dijkstra pour trouver le plus court chemin entre les sommets

```
depart = 401
arrivee = 200
```

- On
  - Affichera le sommet `depart` en Vert
  - et le sommet `arrivee` en Rouge
- On initialise une variable `time_start` à `time.clock()` (ou `time.process_time()` selon les versions de Python, package time) pour prendre le temps :
  - avant l'algorithme
  - et après l'algorithme,
  - pour avoir le temps de calcul en faisant la différence.
- On peaufinera les affichages



Avec Networkx

TP DIJKSTRA

Structures

- NetworkX gère un graphe comme un dictionnaire de dictionnaires de dictionnaires…
- NetworkX permet d'accéder à des informations sur le graphe
  - Dans un graphe orienté `G.successors(j)` retourne le dictionnaire des sommets successeurs de `j`
    - « `for k in G.successors(j):` » permet de traiter tous les successeurs `k` de `j`
  - `G[i][j]['weight']` est le poids (la longueur) de l'arc (i,j)

ATTENTION ! Le graphe est orienté

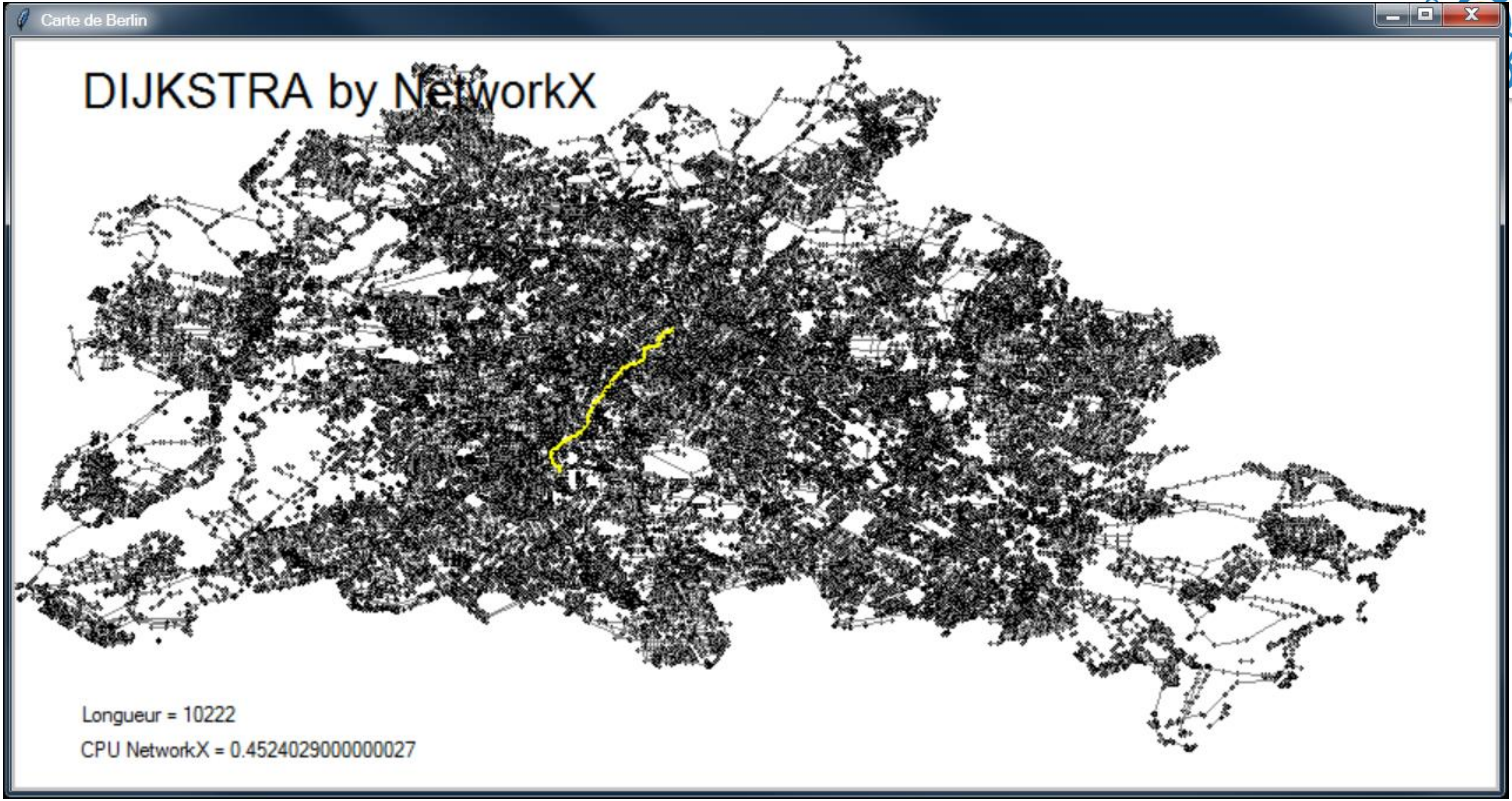
```
G = nx.DiGraph()
nx.set_node_attributes(G,0,"X")
nx.set_node_attributes(G,0,"Y")
```

TP DIJKSTRA

Dijkstra via NetworkX

- Avec NetworkX, la méthode s'appelle `nx.dijkstra_path()`.
- Elle retourne la liste des nœuds qui constituent le plus court chemin.
- Regarder les autres méthodes utilisables avec NetworkX
- Afficher le temps de calcul et le poids trouvé, à l'écran (`print`) mais aussi sur la figure.

```python
# ###########################################
# Appel de la méthode de Networkx
# ###########################################
print('*** nx.dijkstra_path ***')

depart = 401
arrivee = 200

time_start = time.process_time()

Path = nx.dijkstra_path(G,depart,arrivee)

time_end = time.process_time()

Long=0
for j in range(len(Path)-1):
    o = Path[j]
    d = Path[j+1]
    Long = Long + G[o][d]['weight']
    #TraceArc(o,d,'yellow',2)

#print('Path=',Path)
cpu_networkx = time_end-time_start

print("Chemin by nx.dijkstra_path : Long =",Long,"CPU =",cpu_networkx)
```


Carte de Berlin
DIJKSTRA by NetworkX
Longueur = 10222
CPU NetworkX = 0.4524029000000027

« A la main »

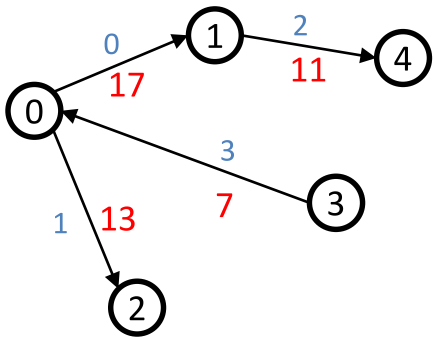
MyCode

TP DIJKSTRA

Prec, Succ, ArcSucc…

- On introduit les structures de données suivantes :
  - `Prec` est une liste qui contient des listes : `Prec[j]` est la liste des prédécesseurs de `j`
  - `Succ` est une liste qui contient des listes : `Succ[j]` est la liste des successeurs de `j`
  - `ArcSucc` est une liste qui contient des listes : `ArcSucc[j]` est la liste des numéros des arcs successeurs de `j`
  - `LongSucc` est une liste qui contient des listes : `LongSucc[j]` est la liste des longueurs des arcs successeurs de `j`

- Exemple




```
Prec = [[3],[0],[0],[],[1]]
Succ = [[1,2],[4],[],[0],[]]
ArcSucc = [[0,1],[2],[],[3],[]]
LongSucc = [[17,13],[11],[],[7],[]]
```

- Initialiser ces listes à vide
- Remplir ces listes

TP Dijkstra

Prec, Succ, ArcSucc…

```python
# Construction des structures de données
Prec = [[] for j in range(NbNoeuds)]
Succ = [[] for j in range(NbNoeuds)]
Long_Succ = [[] for j in range(NbNoeuds)]
Long_Prec = [[] for j in range(NbNoeuds)]
ArcSucc = [[] for j in range(NbNoeuds)]
ArcPrec = [[] for j in range(NbNoeuds)]
for u in range(NbArcs):
    i = Origine[u]
    j = Destination[u]
    Succ[i].append(j)
    Prec[j].append(i)
    Long_Succ[i].append(Longueur[u])
    Long_Prec[j].append(Longueur[u])
    ArcSucc[i].append(u)
    ArcPrec[j].append(u)
```

TP DIJKSTRA

Dijkstra

Algorithme du cours

Comment on l'implémente

MyCode

- Implémenter Dijkstra

- On utilisera :
  - Une liste `Pi`, <u>initialisée</u> à l'infini pour chaque nœud, qui contiendra le potentiel du nœud
  - Une liste `Pere`, <u>initialisée</u> à -1 pour chaque nœud, qui contiendra le père du nœud `j` dans le plus court chemin de `depart` à `j`
  - Une liste `Candidats` <u>initialisée</u> à vide, qui contiendra tous les sommets candidats de $\bar{S}$, i.e. tous ceux sont atteignables et qui n'ont pas de potentiel définitif.
  - Une liste `Marque` qui permettra de savoir si un sommet a quitté $\bar{S}$. Elle est <u>initialisée</u> à `False` pour tous les nœuds. « Marquer un sommet `j` », c'est mettre `Marque[j]` à `True`.

`Marque[j] = False` $\Leftrightarrow$ `j` $\in \bar{S}$

Dijkstra

\# Initialisation :
$\pi(s) \leftarrow 0, \bar{\pi}(s) \leftarrow 0$
$\bar{S} \leftarrow \{1,2,\dots,N\}\backslash\{s\}$
$$\pi(j) \leftarrow \begin{cases} l_{s,j} & \text{si } (s,j) \in U \\ +\infty & \text{sinon} \end{cases}$$
$$Pere(j) \leftarrow \begin{cases} s & \text{si } (s,j) \in U \\ -1 & \text{sinon} \end{cases}$$
Tantque $\bar{S} \neq \emptyset$ Faire
    Sélectionner $j \in \bar{S}, \pi(j) = \min_{i \in \bar{S}} \pi(i)$
    $\bar{S} \leftarrow \bar{S}\backslash\{j\}$
    $\bar{\pi}(j) \leftarrow \pi(j)$
    Pour tout $(j,k) \in U, k \in \bar{S}$ Faire
        Si $\pi(k) \geq \pi(j) + l_{j,k}$ Alors
            $\pi(k) \leftarrow \pi(j) + l_{j,k}$
            $Pere(k) \leftarrow j$
    FinPour
FinTantque

TP DIJKSTRA

Dijkstra

- Implémenter Dijkstra

1. Initialiser
   - le potentiel `Pi` de `depart` à `0`
   - Marquer `depart` (⇔ retirer `depart` de $\bar{S}$)
   - **Pour** chaque successeur `j` de `depart` , **Faire**
     - initialiser le potentiel `Pi` de `j` à la longueur de l'arc (`depart,j`)
     - Initialiser le `Pere` de `j` à `depart`
     - Ajouter `j` à la liste `Candidats`
   - **Finpour**

MyCode

Dijkstra

\# Initialisation :

$\pi(s) \leftarrow 0, \bar{\pi}(s) \leftarrow 0$

$\bar{S} \leftarrow \{1,2,\dots,N\} \backslash \{s\}$

$\pi(j) \leftarrow \begin{cases} l_{s,j} & \text{si } (s,j) \in U \\ +\infty & \text{sinon} \end{cases}$

$Pere(j) \leftarrow \begin{cases} s & \text{si } (s,j) \in U \\ -1 & \text{sinon} \end{cases}$

Tantque $\bar{S} \neq \emptyset$ Faire

Sélectionner $j \in \bar{S}, \pi(j) = \min_{i \in \bar{S}} \pi(i)$

$\bar{S} \leftarrow \bar{S} \backslash \{j\}$

$\bar{\pi}(j) \leftarrow \pi(j)$

Pour tout $(j,k) \in U, k \in \bar{S}$ Faire

Si $\pi(k) \geq \pi(j) + l_{j,k}$ Alors

$\pi(k) \leftarrow \pi(j) + l_{j,k}$

$Pere(k) \leftarrow j$

FinPour

FinTantque

TP DIJKSTRA

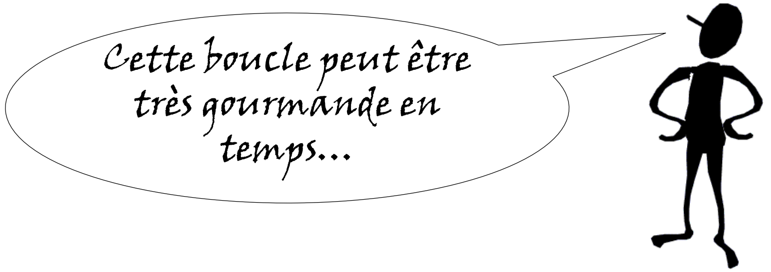
MyCode

Dijkstra

- Implémenter Dijkstra

1. Initialiser un booléen `fini` à `Faux` (ce sera fini lorsque le nœud `arrivee` sera atteint)
2. **Tant que** (la liste `Candidats` n'est pas vide) et (pas `fini`) **Faire**
    1. Parcourir tous les sommets de la liste **Candidats** et parmi ceux qui ne sont pas marqués, trouver celui qui a le plus petit potentiel, noté `noeud_retenu`.
    2. Marquer ce `noeud_retenu`
    3. Afficher ce `noeud_retenu` en beige par `TraceCercle(noeud_retenu ,'beige',1)`
    4. …

…(suite au dos)



Dijkstra

\# Initialisation :

$\pi(s) \leftarrow 0, \bar{\pi}(s) \leftarrow 0$

$\bar{S} \leftarrow \{1,2,\dots,N\}\backslash\{s\}$

$$\pi(j) \leftarrow \begin{cases} l_{s,j} & \text{si } (s,j) \in U \\ +\infty & \text{sinon} \end{cases}$$

$$Pere(j) \leftarrow \begin{cases} s & \text{si } (s,j) \in U \\ -1 & \text{sinon} \end{cases}$$

Tantque $\bar{S} \neq \emptyset$ Faire

Sélectionner $j \in \bar{S}, \pi(j) = \min_{i \in \bar{S}} \pi(i)$

$\bar{S} \leftarrow \bar{S}\backslash\{j\}$

$\bar{\pi}(j) \leftarrow \pi(j)$

Pour tout $(j,k) \in U, k \in \bar{S}$ Faire

Si $\pi(k) \geq \pi(j) + l_{j,k}$ Alors

$\pi(k) \leftarrow \pi(j) + l_{j,k}$

$Pere(k) \leftarrow j$

FinPour

FinTantque

TP DIJKSTRA

Dijkstra

- Implémenter Dijkstra

1. **Tant que** (la liste des `candidats` n'est pas vide) et (pas `fini`) **Faire**
   3. …
   4. **Si** `noeud_retenu` = `arrivee` **Alors** mettre `fini` à `Vrai` **Finsi**
   5. **Pour** tout successeur `k` de `noeud_retenu` , `k` non marqué **Faire**
      1. Calculer le potentiel de `k` en passant par `noeud_retenu`
      2. si besoin (donc Si $\pi(k) \geq \pi(noeud\_retenu) + l_{\text{noeud\_retenu},k}$) :
         1. Mettre à jour le potentiel de `k`
         2. Mémoriser le nouveau `Pere` (qui sera `noeud_retenu`)
         3. Insérer `k` dans la liste des `Candidats`

   Finpour

FinTantQue

Dijkstra

\# Initialisation :

$\pi(s) \leftarrow 0, \bar{\pi}(s) \leftarrow 0$

$\bar{S} \leftarrow \{1,2,\ldots,N\} \backslash \{s\}$

$\pi(j) \leftarrow \begin{cases} l_{s,j} & \text{si } (s,j) \in U \\ +\infty & \text{sinon} \end{cases}$

$Pere(j) \leftarrow \begin{cases} s & \text{si } (s,j) \in U \\ -1 & \text{sinon} \end{cases}$

Tantque $\bar{S} \neq \emptyset$ Faire

Sélectionner $j \in \bar{S}, \pi(j) = \min_{i \in \bar{S}} \pi(i)$

$\bar{S} \leftarrow \bar{S} \backslash \{j\}$

$\bar{\pi}(j) \leftarrow \pi(j)$

Pour tout $(j,k) \in U, k \in \bar{S}$ Faire

Si $\pi(k) \geq \pi(j) + l_{j,k}$ Alors

$\pi(k) \leftarrow \pi(j) + l_{j,k}$

$Pere(k) \leftarrow j$

FinPour

FinTantque

TP DIJKSTRA

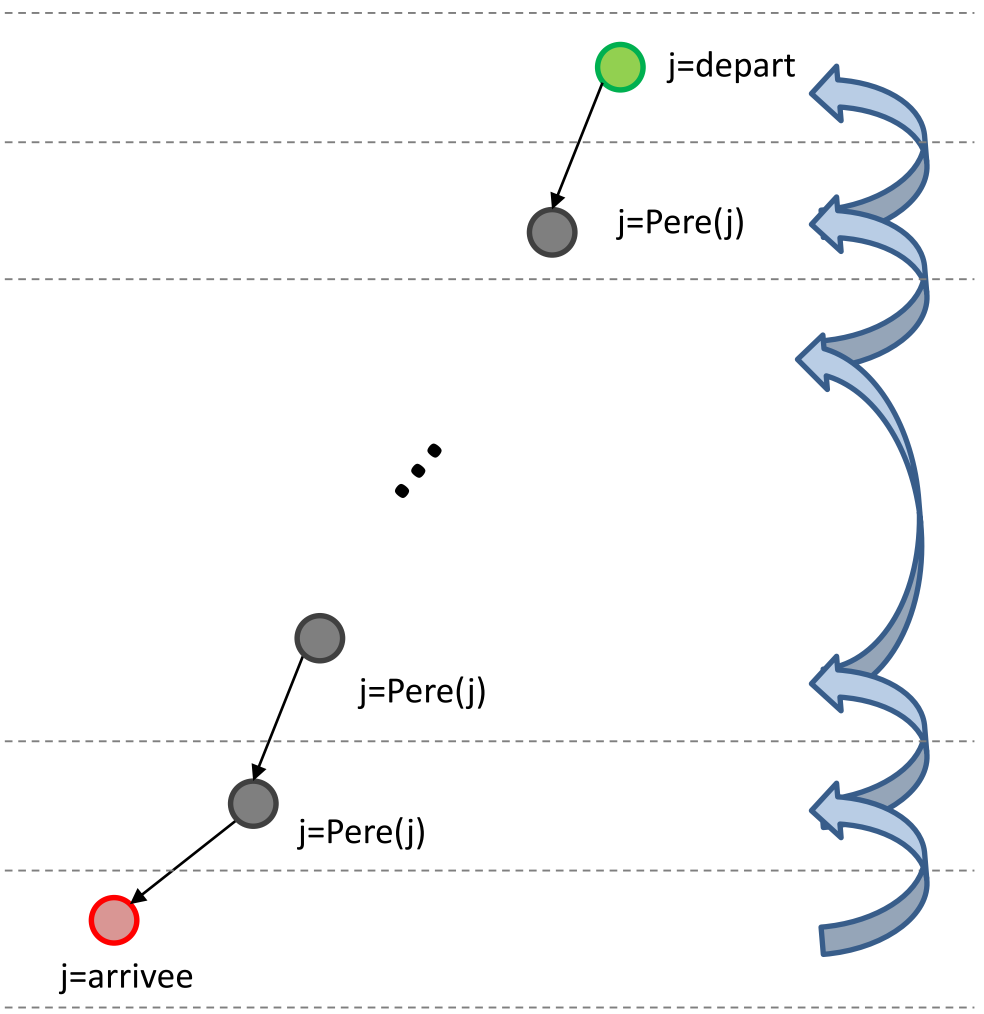
MyCode

Le chemin

Ensuite, il s'agit d'identifier les arcs qui constituent le chemin trouvé

- Ecrire la procédure de backtrack qui utilise la liste `Pere`
- La longueur du chemin est donnée par `Pi[arrivee]`
- Afficher chaque sommet qui se trouve sur le chemin en utilisant `TraceCercle (j,'red',2)`
- Afficher le temps de calcul de la routine.

- Tester avec d'autres sommets `depart` et `arrivee`



TP DIJKSTRA

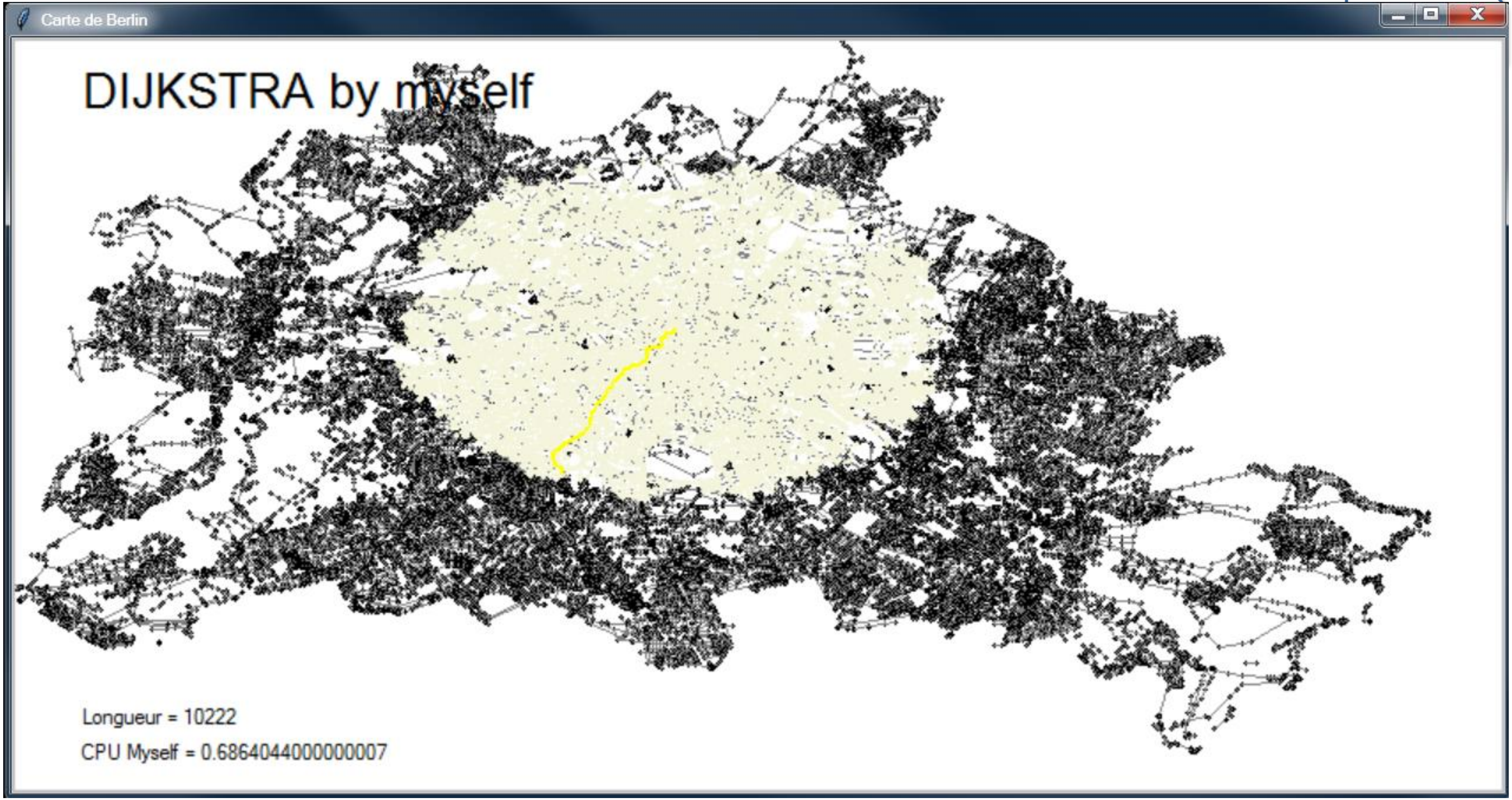
MyCode

Amélioration

> Parcourir tous les sommets de la liste **Candidats** et parmi ceux qui ne sont pas marqués, trouver celui qui a le plus petit potentiel, noté `noeud_retenu`.

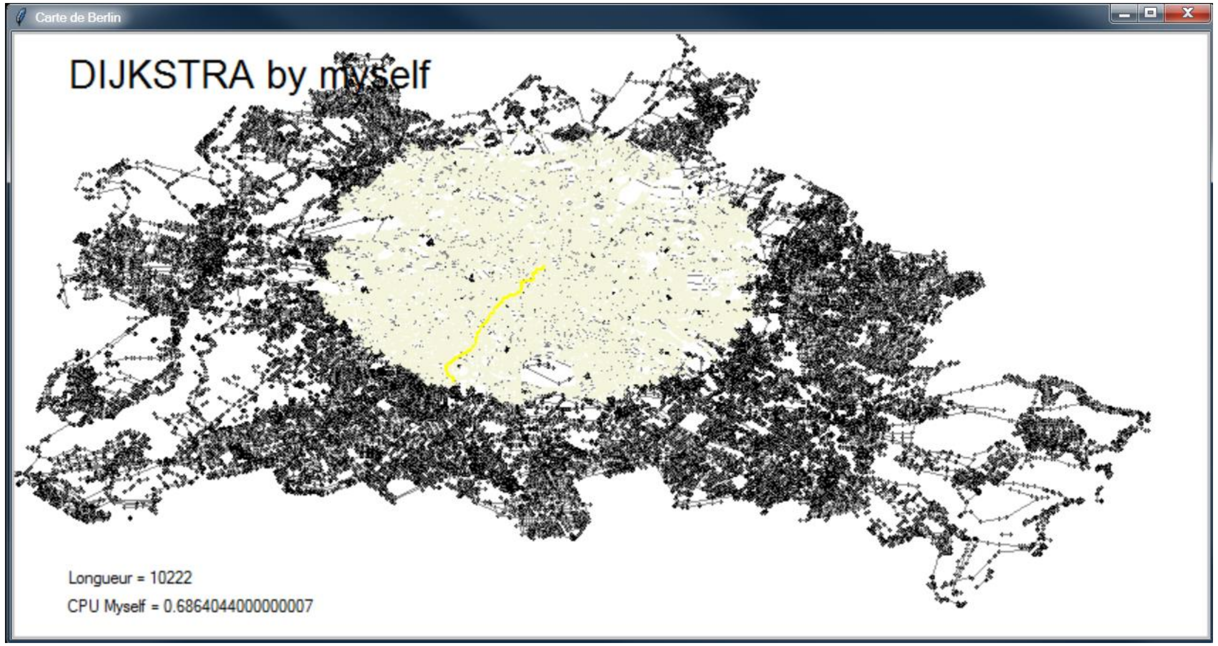
- Cette partie du code peut prendre un grand temps de calcul… surtout s'il y a beaucoup de sommets dans le graphe.
- Alternative : avoir une liste des candidats triée par potentiels croissants
- Idée :
  - Écrire une procédure qui insère un nœud dans la liste des candidats en fonction de son potentiel. On peut chercher la position d'insertion (éventuellement par dichotomie) et appeler `Liste.insert(pos,élément)`
  - Puis systématiquement, prendre le 1er sommet de la liste.

Faire cette partie est obligatoire avant de passer à l'algorithme A★


Carte de Berlin
DIJKSTRA by myself
Longueur = 10222
CPU Myself = 0.6864044000000007

TP A☆ (lire A-étoile ou A-star)

- L'algorithme de Dijkstra explore les sommets en formant une « boule ».
- En effet, à chaque itération, on prend le sommet le plus proche du sommet de départ (celui qui a le plus petit potentiel).
- L'idée de A☆ est d'explorer en priorité les sommets les plus proches de l'arrivée.
- Pour cela, on va calculer une distance « à vol d'oiseau » d'un sommet à l'arrivée.



Avec Networkx

TP A☆

A☆ via NetworkX

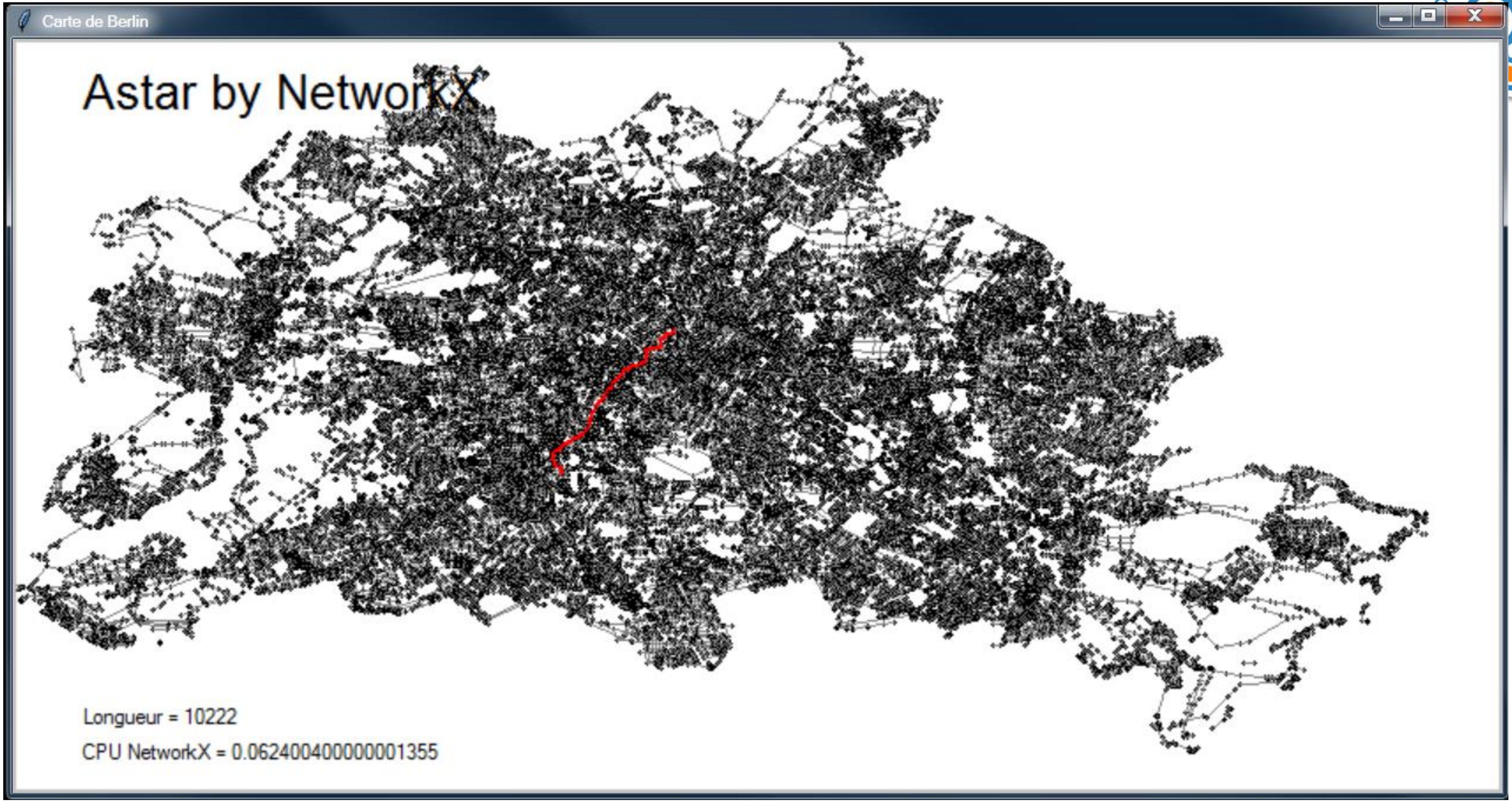
- Avec NetworkX, la méthode s'appelle **astar_path**().
- Elle prend comme paramètre une fonction qui calcule la distance entre deux sommets.
- Elle retourne la liste des nœuds qui constituent le plus court chemin.
- Afficher le temps de calcul et le poids trouvé, à l'écran (print) mais aussi sur la figure.

```python
# ########################################
# Appel de la méthode de Networkx
# ########################################
print('*** astar_path ***')

depart = 401
arrivee = 200

def Distance_vol_oiseau_networkX (villeA,villeB):
```

Voir plus loin

```python
time_start = time.process_time()

Path = nx.

time_end = time.process_time()
```

```python
#print('Path=',Path)
cpu_networkx = time_end-time_start
```


Carte de Berlin
Astar by NetworkX
Longueur = 10222
CPU NetworkX = 0.062400400000001355

« A la main »

MyCode

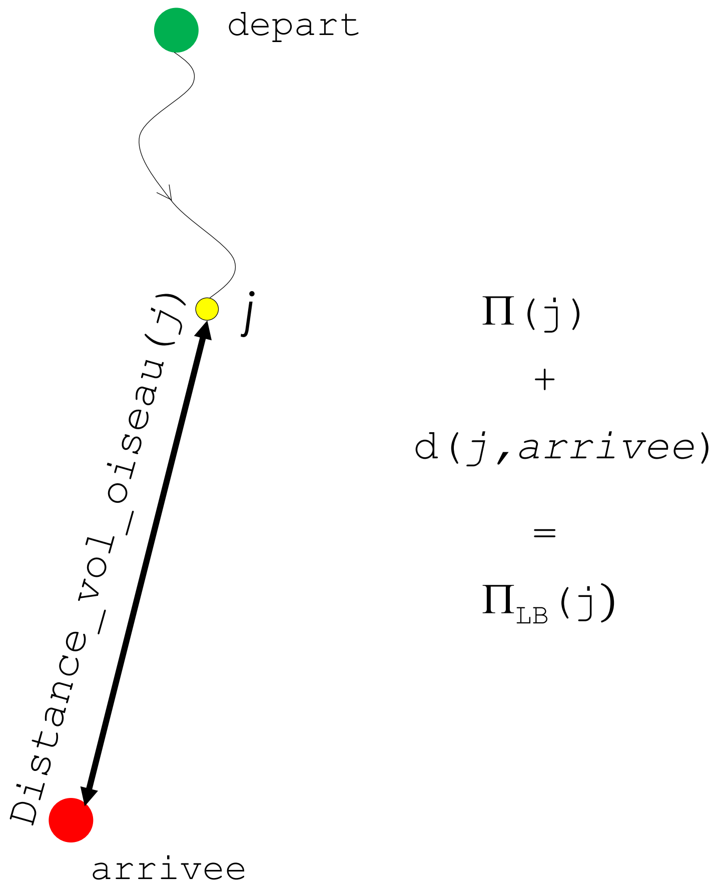
TP A☆

```
coeff = 70
def Distance_vol_oiseau (villeA):
    # calcule la distance de la ville A a la destination
    xA = X[villeA]
    xB = X[arrivee]
    yA = Y[villeA]
    yB = Y[arrivee]
    d = coeff*sqrt((xA-xB)**2+(yA-yB)**2)
    return(d)
```

- Le principe de A☆ est le suivant.
  - Pour chaque sommet candidat, on calcule sa distance à `arrivee` par la procédure `Distance_vol_oiseau`
  - On a donc pour chaque sommet candidat `j` une distance à `arrivee` et une estimation (une sous-estimation car en pratique ce sera plus long) de la distance finale entre `depart` et `arrivee` en faisant :

$$\Pi_{LB}(j) = \Pi(j) + d(j,arrivee)$$

  - On crée une nouvelle liste `PiLB_trie` qui contient ces estimations dans l'ordre croissant.
  - On positionne `j` dans la liste des `Candidats` à la même place que l'on positionne sa borne inférieure dans `PiLB_trie`.
    - Donc c'est presque pareil que la méthode suggérée avec le tri, sauf que l'on se base sur `Pi[j]+d(j,arrivée)` pour trier



TP A☆

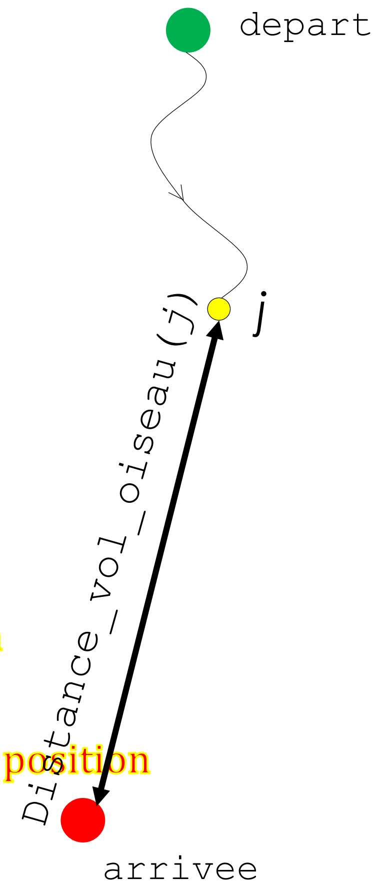
MyCode

Algorithme A☆

```
Ecrire une fonction qui retourne
la position d'insertion
Ou
Utiliser le package Python « bisect »
```

Initialisations

- Pi = [∞, ∞, … , ∞] ∀j
- PiLB = [∞, ∞, … , ∞] ∀j
- LePrec = [-1, -1, … , -1] ∀j
- Marque = [0, 0, … , 0] ∀j
- Candidats = []
- PiLB_Trie = []
- Pi[sommet_depart] = 0
- Marque[sommet_depart] = 1
- **Pour** tous les successeurs k de sommet_depart **Faire**
  - Pi[k] = longueur de l'arc (sommet_depart,k)
  - LePrec[k] = sommet_depart
  - Calculer PiLB[k], potentiel de k + distance à vol d'oiseau de k à sommet_destination
  - Déterminer la position d'insertion de PiLB[k] dans PiLB_Trie
  - Insérer PiLB[k] dans cette liste à cette position, insérer k dans Candidats à la même position
- **FinPour**



$$\Pi(j) + d(j, arrivee) = \Pi_{LB}(j)$$

TP A☆

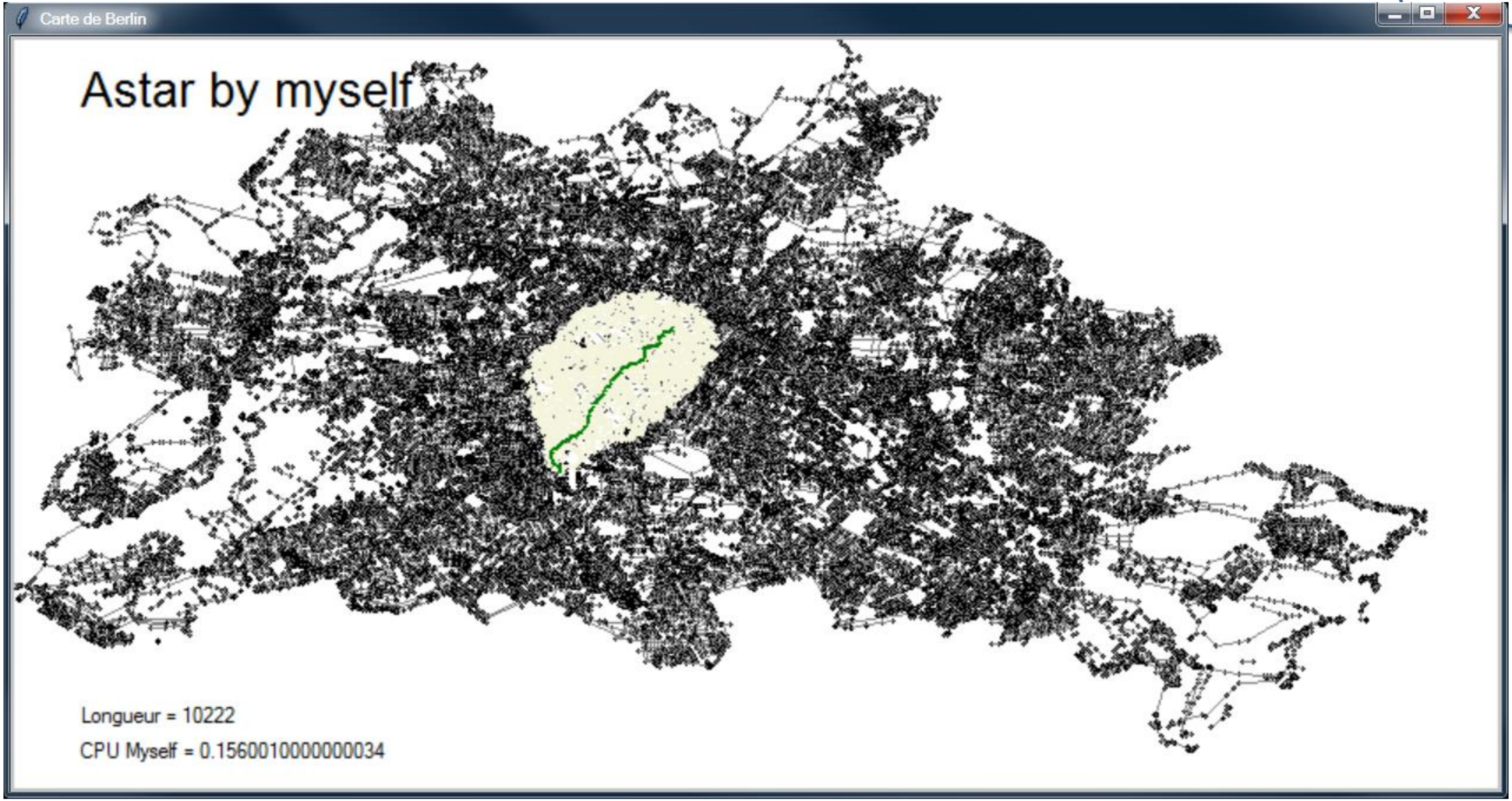
Algorithme A☆

- Fini ← Faux
- **Tant que** not Fini **Faire**
  - Prendre le 1er élément de Candidats noté j
  - Marquer ce sommet j, le tracer en jaune, le retirer de Candidats, retirer son potentiel de la liste PiLB_trie
  - **Si** j = sommet_destination **Alors** Fini ← Vrai
  - **Pour** chaque successeur k de j **Faire**
    - **Si** k n'est pas marqué **Alors**
      - Calculer le nouveau potentiel de k Pi[k] en passant par j
      - **Si** le nouveau potentiel est inférieur à l'actuel **Alors**
        - Chercher k dans la liste des Candidats et (s'il y est)le retirer, retirer aussi son potentiel de PiLB_trie
        - Mettre à jour le potentiel de k, le père de k
        - Calculer la borne inférieure PiLB[k] du sommet k
        - Ajouter cette borne inférieure dans la liste PiLB_trie, à sa place
        - Ajouter k aux Candidats à la même place
      - **Finsi**
    - **Finsi**
  - **FinPour**
- **FinTQ**

MyCode

TP A☆

MyCode



Carte de Berlin
Astar by myself
Longueur = 10222
CPU Myself = 0.1560010000000034