TIPE 2018-2019

Thème : Transport

Filière MP option SI

Transport du phosphate en Tunisie

Objectif : Tenter d'optimiser le transport du phosphate à travers un réseau de pipeline .

Transport du phosphate par train

Transport du phosphate par pipeline

Comment optimiser les coûts du transport du phosphate dans un réseau de pipeline ?

Les attendus de mon projet

- ❖Réalisation d'une étude physique.
- ❖Modélisation du trajectoire par un réseau avec la théorie des graphes.
- ❖Application de l'algorithme de Ford Fulkerson et l'algorithme de Dijkstra.
- ❖Comparer avec OCP au Maroc .

Sommaire

- I - Introduction
- II – Modélisation physique :
  1. Equation de Bernoulli.
  2. Résolution numérique de l'équation du perte de charge.
- III - Modélisation mathématique :
  1. Modélisation du circuit de slurry pipeline par un graphe pondéré.
  2. Explication de quelques algorithmes et les exécutés.
- IV - Conclusion.

Contributions

Janvier 2019: Visite de l’usine TIFERT au cours de laquelle j’ai contacté monsieur Ramzi Hmidi directeur de l’usine.

Cet usine est parmi les grandes manufactures concernant la production de l’acide phosphorique.

Unité de production

Pompe centrifuge

Problématique: Comment assurer un transport optimal de phosphates?

Caractéristiques du phosphate

Phosphate + Soufre

Joseph Pelletier: Pharmacien et chimiste français

Pierre Louis Dulong: chimiste et physicien français

Humphry Davy: chimiste et physicien britannique

Étudier ses divers acides

Différents types de transport du phosphate en Tunisie

Transport ferroviaire

Transport par camion

Les limitations

les frais de transport sont trop élevés

5DT(train)

20 DT(camion)

les soulèvements sociaux

Méthodologie adoptée

Protection des matériaux

Étudier le nombre de Reynolds

Méthodologie adoptée

Protection des matériaux

Étudier le nombre de Reynolds

Facteur de perte de charge

Formule de colebrook

Méthodologie adoptée

- Protection des matériaux
- Étudier le nombre de Reynolds
- Facteur de perte de charge
- Formule de colebrook
- Équation de Bernoulli

Méthodologie adoptée

Protection des matériaux

Étudier le nombre de Reynolds

Facteur de perte de charge

Formule de colebrook

Équation de Bernoulli

Étude longitudinale

Calcul de perte de charge

Les stations de pompage

Méthodologie adoptée

Protection des matériaux

Étudier le nombre de Reynolds

Facteur de perte de charge

Formule de colebrook

Équation de Bernoulli

Étude longitudinale

Calcul de perte de charge

Les stations de pompage

Modélisation du trajet

Méthodologie adoptée

Protection des matériaux

Étudier le nombre de Reynolds

Facteur de perte de charge

Formule de colebrook

Équation de Bernoulli

Étude longitudinale

Calcul de perte de charge

Les stations de pompage

Modélisation du trajet

Matrice d’adjacence

Application de l’algorithme de Dijkstra

Calcul de complexité

Méthodologie adoptée

Protection des matériaux

Étudier le nombre de Reynolds

Facteur de perte de charge

Formule de colebrook

Équation de Bernoulli

Étude longitudinale

Calcul de perte de charge

Les stations de pompage

Modélisation du trajet

Matrice d’adjacence

Application de l’algorithme de Dijkstra

Calcul de complexité

Matrice d’adjacence

Application de l’algorithme ford fulkerson

Mais comment peut on éviter les accidents tout au long le pipeline ?

La composition du fluide transporté

Phosphate + eau → Fluide homogène

Solution → Avoir un régime turbulent pour éviter la sédimentation dans la canalisation → $R_e \geq 3000$

Calculer le nombre de Reynolds

$$R_e = \frac{VD}{v} = \frac{\rho VD}{\mu}$$

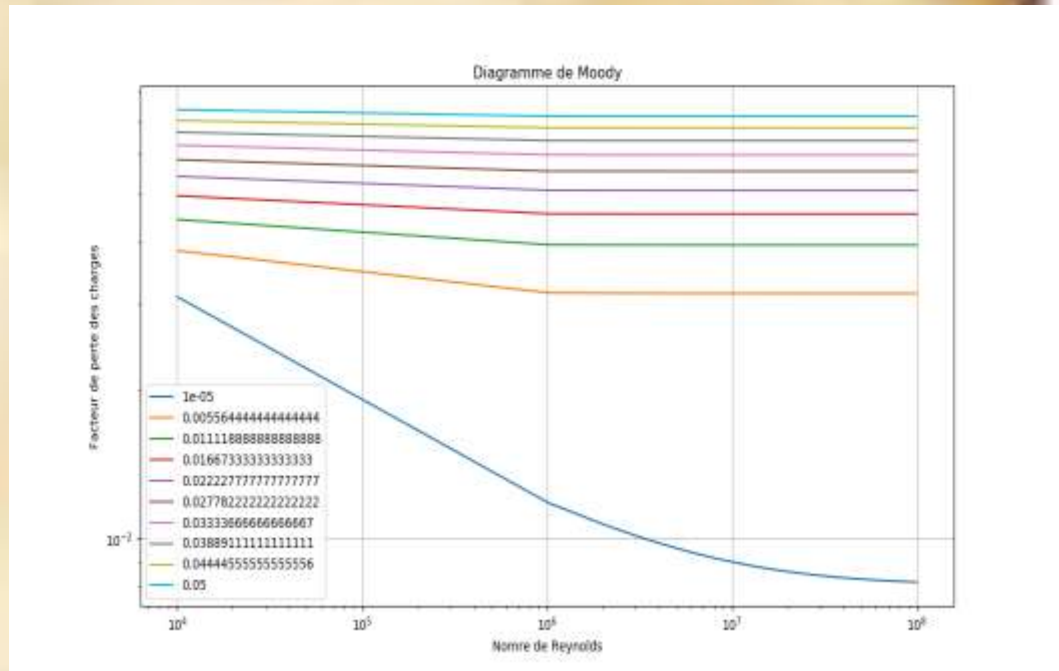
Ou peut on placer les stations de pompage ?

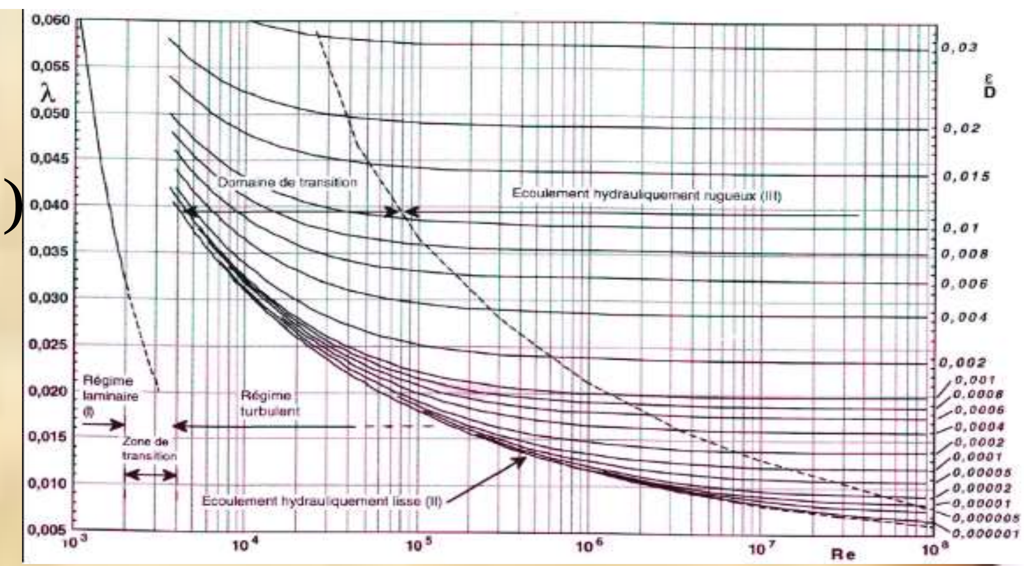
Nous sommes dans le cas de régime turbulent.

Utilisation de formule de colebrook

$$\frac{1}{\sqrt{\lambda}} = -2\log\left[\frac{k}{3.7} + \frac{2.51}{R_e\sqrt{\lambda}}\right]$$

Simulation numérique $\lambda = f(R_e)$ et le comparé avec le diagramme de Moody (diagramme expérimental)





Je vais simplifier au mieux les hypothèses et approximations sur le fluide pour que l'exercice de modélisation soit dans mes capacités

- Un fluide incompressible
- Un fluide parfait
- En régime stationnaire
- On néglige les transferts d'énergie sous forme de chaleur

Equation de Bernoulli généralisée

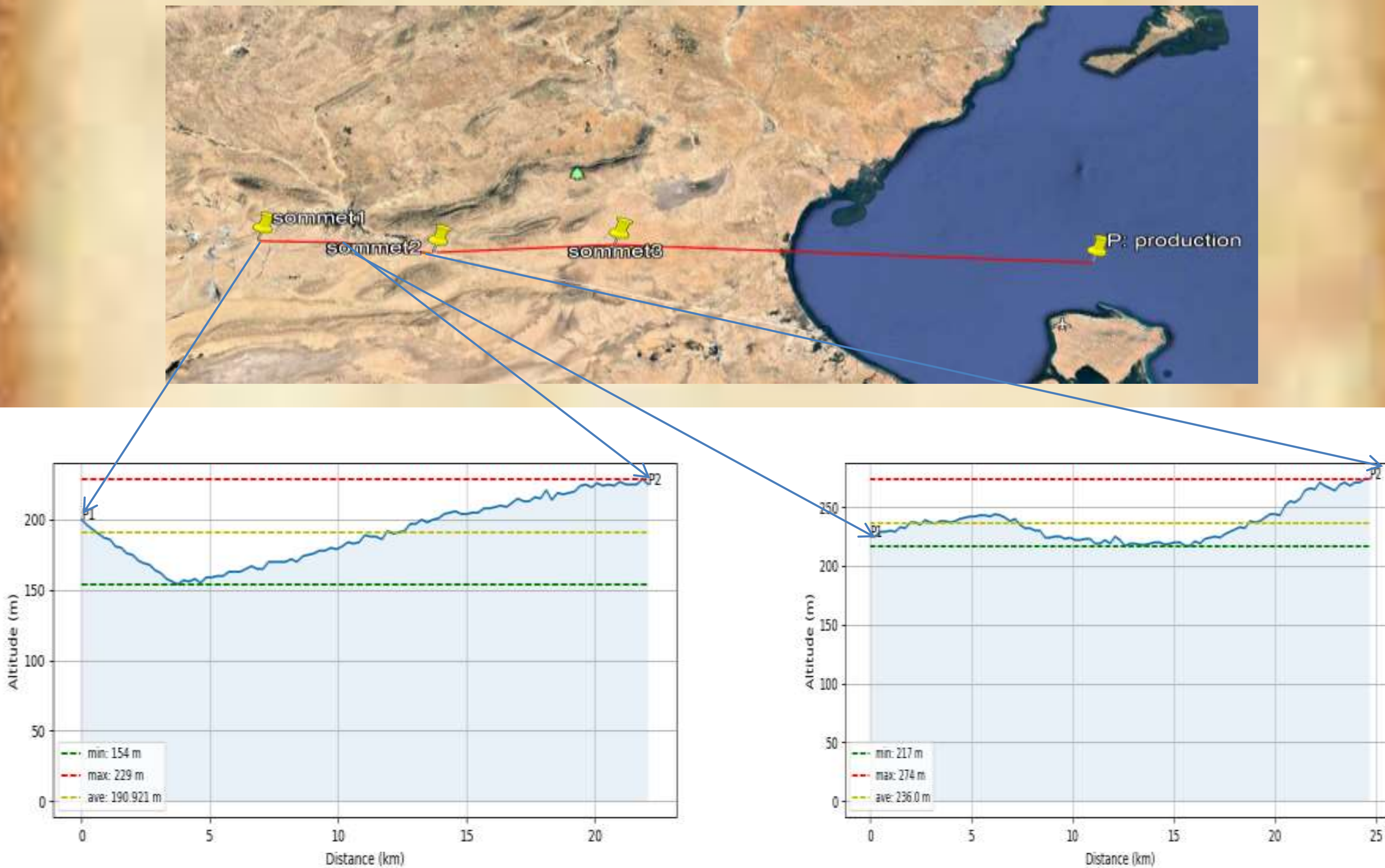
$$P_1 + \rho g z_1 + \rho \frac{v_1^2}{2} = p_2 + \rho g z_2 + \rho \frac{v_2^2}{2} + \Delta p_f$$

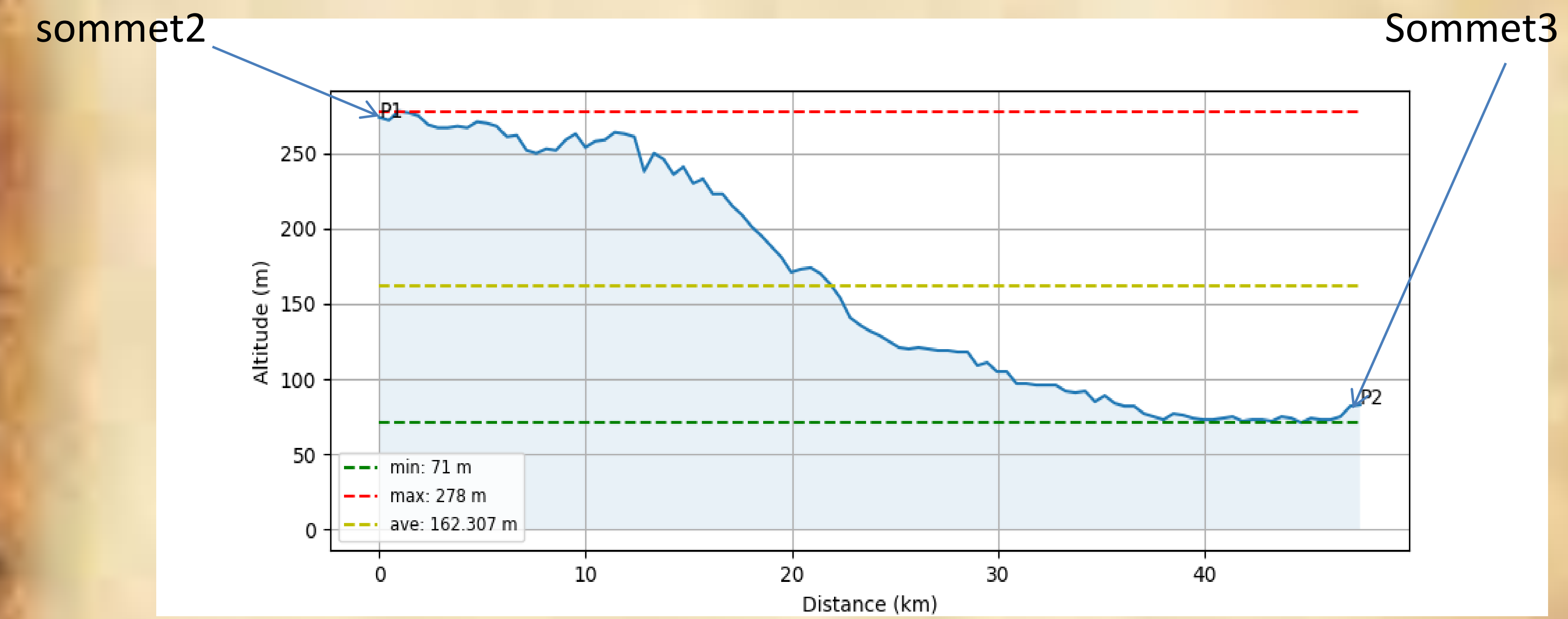
$$\Downarrow$$

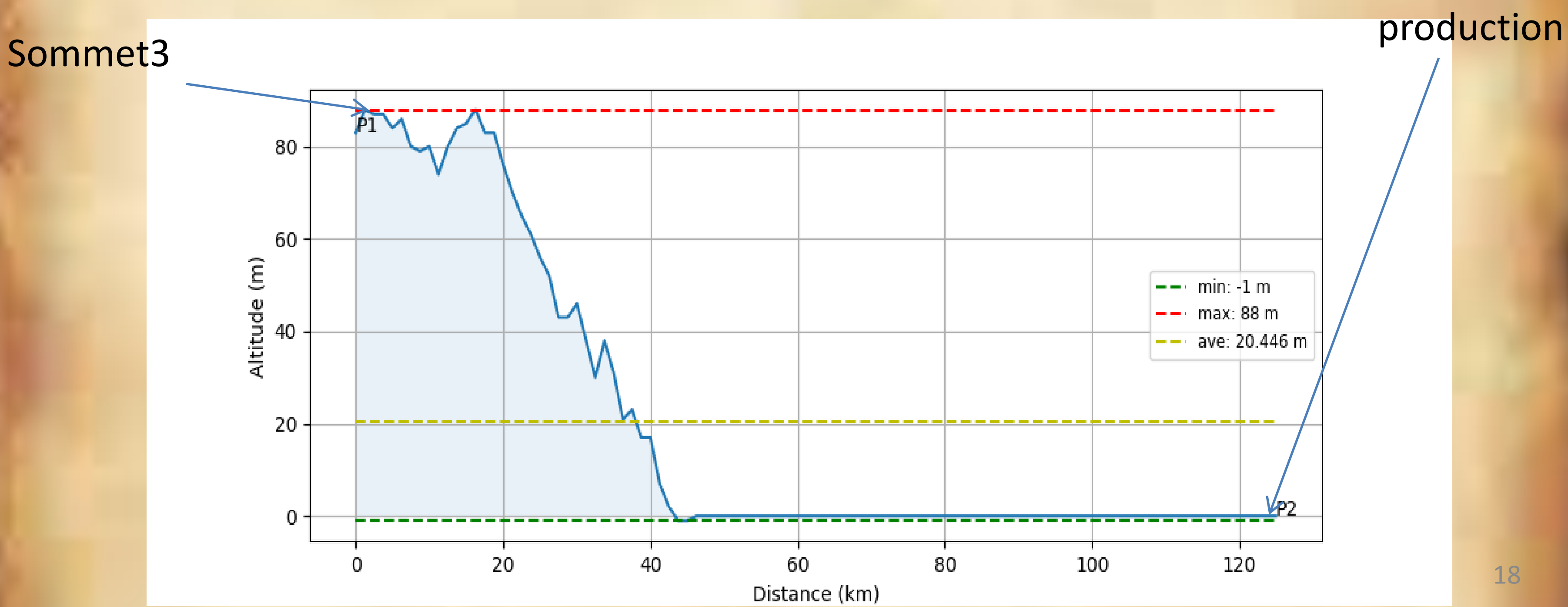
$$h_1 + z_1 + \frac{v_1^2}{2g} = h_2 + z_2 + \frac{v_2^2}{2g} + \Delta h_f$$

Comment peut on étudier le comportement du pipeline ?

Réalisation d’une étude longitudinale tout au long le pipeline




sommet2
Sommet3
P1
P2
min: 71 m
max: 278 m
ave: 162.307 m
Altitude (m)
Distance (km)


Sommet3
production
P1
P2
min: -1 m
max: 88 m
ave: 20.446 m
Altitude (m)
Distance (km)

La perte de charge

$$\Delta p = \lambda \frac{L}{D} \frac{\rho V^2}{2}$$

On a

$$P_1 + \rho g z_1 + \rho \frac{V_1^2}{2} = p_2 + \rho g z_2 + \rho \frac{V_2^2}{2} + \Delta p_f$$

Hypothèses :

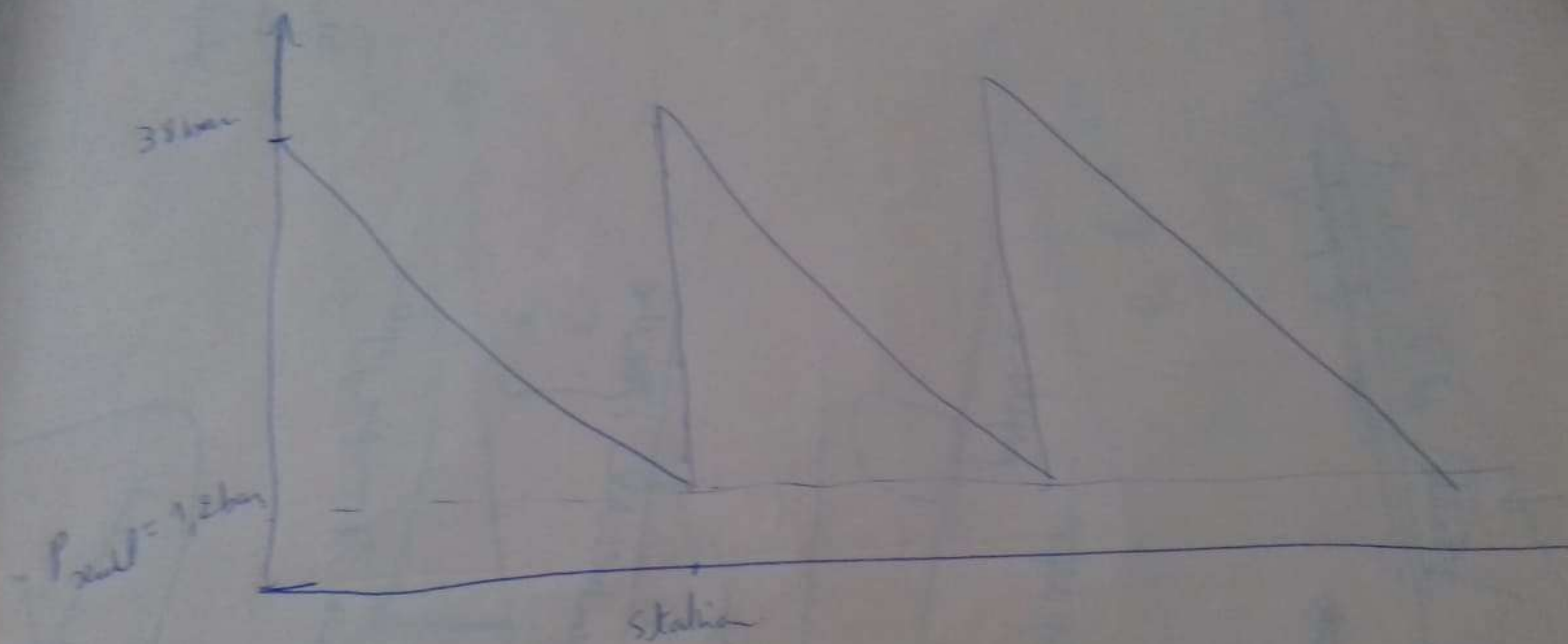
$$\begin{cases} Z_1 = Z_2 \\ V_1 = V_2 \end{cases}$$

$$P_1 = P_2 + \Delta P_f \quad \Rightarrow \quad P_1 = P_2 + \Delta P$$

Avec $\Delta P = \Delta P_f$

$$\Rightarrow L = \frac{2(P_1 - P_2)D}{\lambda \rho V^2}$$

Détermination des positions des stations de pompage


P seuil = 1,2 bar
station

Modélisation

La trajectoire_réelle du pipeline

La modélisation est une étape essentielle pour comprendre un phénomène en utilisant des différentes outils

Les outils :

- l'attitude
- longitude
- capacités

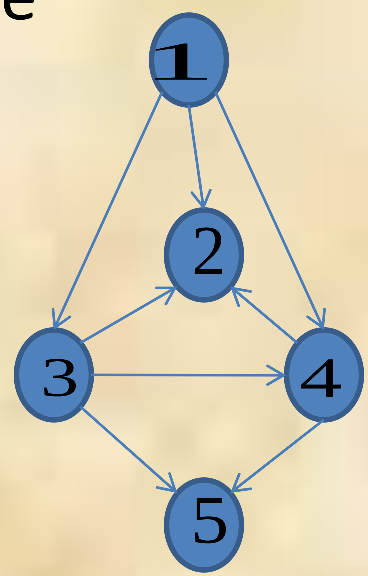
distance

Comment peut-on modéliser ce problème ?

Solution: on modélise ce problème par la théorie de graphe

Les graphes font partie des mathématiques discrètes il est constitué de deux ensembles (ensemble de sommets, ensemble d'arêtes) avec une pondération .

on peut modéliser une carte géographique par une matrice d’adjacence.



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

On peut modélisé la trajectoire réelle du pipeline par le graphe suivant

Utilisation Google Earth pour modéliser le pipeline par un graphe orienté et pondéré telle que la pondération sont les distances qui séparent les sommets.

	A	B	C	D	E	F	G	H
A	0	47,1	0	87	0	105	0	0
B	0	0	31,5	45	56	61	0	0
C	0	0	0	0	32,2	33.3	0	0
D	0	0	0	0	21,7	0	74,3	0
E	0	0	0	0	0	0	64,7	73,8
F	0	0	0	0	0	0	0	73,8
G	0	0	0	0	0	0	0	73,8
H	0	0	0	0	0	0	0	0

Comment déterminer le plus court chemin entre le sommet 1 et le site de production ?

On peut trouver le plus court chemin à travers l'algorithme de Dijkstra réalisé par le mathématicien et l'informaticien néerlandais Edsger Dijkstra.

G=(S,A) un graphe avec une pondération positive poids des arcs, s un sommet de S.

P:=[]

d[a]:=infinity pour chaque sommet a

d[s]=0

INITIALISATIONS

Tant que tout les sommets ne sont pas dans P

choisir un sommet a n'est pas dans P de plus petite distance d[a] ,mettre a dans P

pour chaque sommet b hors de P voisin de a

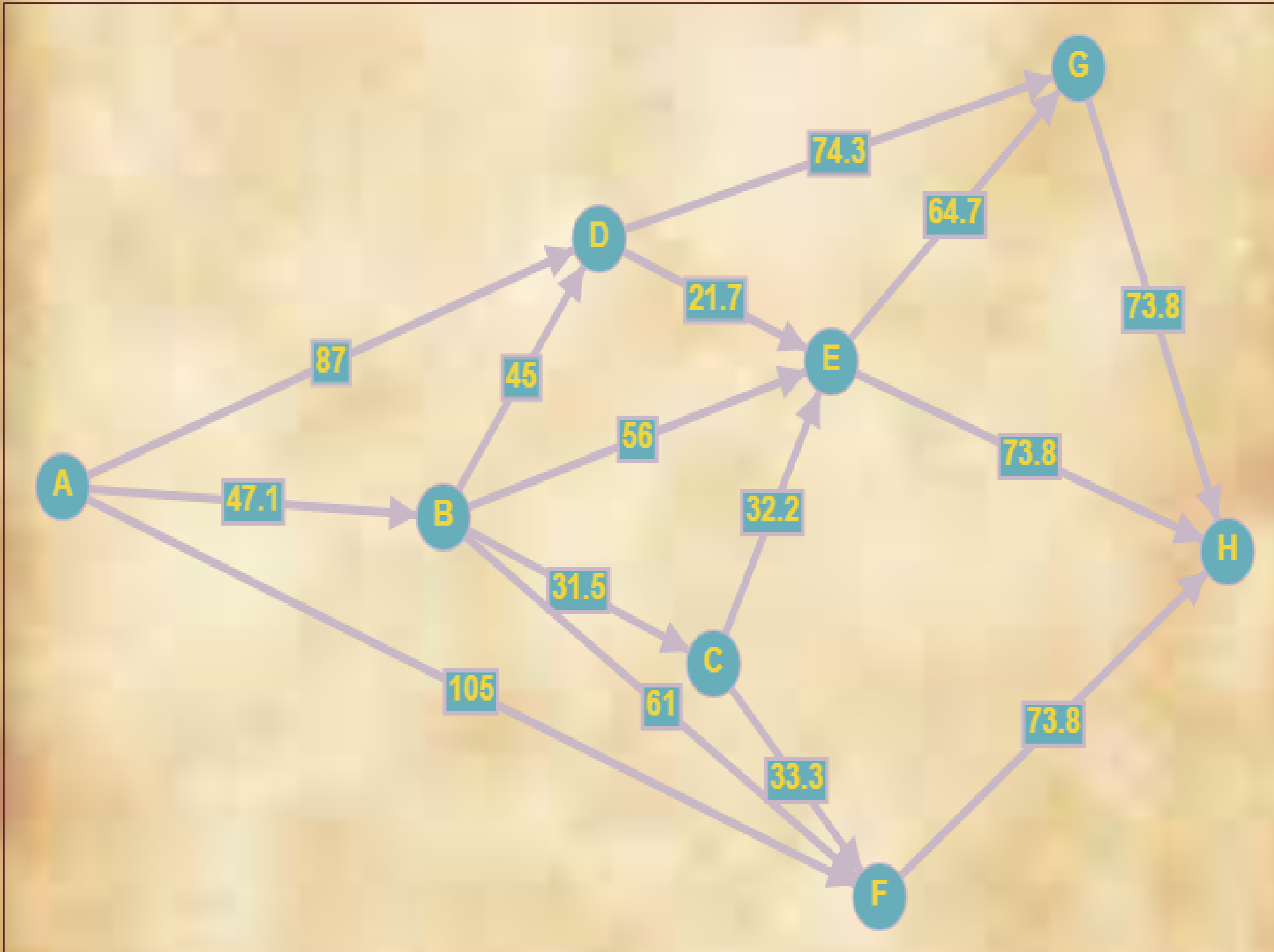
d[a]=min(d[b],d[a] + poids (a,b))

fin pour

Fin tant que

Boucle de traitements

comment ça fonctionne l'algorithme de Dijkstra ?


G
74.3
64.7
D
73.8
21.7
E
87
45
56
73.8
A
47.1
B
32.2
H
31.5
C
105
61
73.8
33.3
F

Algorithme de Dijkstra
P=[]

Sommet définitivement fixé	A	B	C	D	E	F	G	H
A	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$

P=[A]

Algorithme de Dijkstra

P=[A]

Sommet définitivement fixé	A	B	C	D	E	F	G	H
A	0	+∞	+∞	+∞	+∞	+∞	+∞	+∞
B		47.1(A)	+∞	87(A)	+∞	105(A)	+∞	+∞

P=[A,B]

Algorithme de Dijkstra

P=[A,B]

Sommet définitivement fixé	A	B	C	D	E	F	G	H
A	0	+∞	+∞	+∞	+∞	+∞	+∞	+∞
B		47.1(A)	+∞	87(A)	+∞	105(A)	+∞	+∞
C			78.6(B)	92.1(B)	103.1(B)	108.1(B)	+∞	+∞

P=[A,B,C]

Algorithme de Dijkstra

P=[A,B,C]

Sommet définitivement fixé	A	B	C	D	E	F	G	H
A	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
B		47.1(A)	$+\infty$	87(A)	$+\infty$	105(A)	$+\infty$	$+\infty$
C			78.6(B)	92.1(B)	103.1(B)	108.1(B)	$+\infty$	$+\infty$
D				92.1(B)	108.7(B)	108.1(B)	161.3(D)	$+\infty$

P=[A,B,C,D]

Algorithme de Dijkstra
P=[A,B,C,D]

Sommet définitivement fixé	A	B	C	D	E	F	G	H
A	0	+∞	+∞	+∞	+∞	+∞	+∞	+∞
B		47.1(A)	+∞	87(A)	+∞	105(A)	+∞	+∞
C			78.6(B)	92.1(B)	103.1(B)	108.1(B)	+∞	+∞
D				92.1(B)	108.7(B)	108.1(B)	161.3(D)	+∞
E					108.7(D)	108.1(B)	161.3(D)	+∞

P=[A,B,C,D,E]

# Algorithme de Dijkstra
P=[A,B,C,D,E]

Sommet définitivement fixé	A	B	C	D	E	F	G	H
A	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
B		47.1(A)	$+\infty$	87(A)	$+\infty$	105(A)	$+\infty$	$+\infty$
C			78.6(B)	92.1(B)	103.1(B)	108.1(B)	$+\infty$	$+\infty$
D				92.1(B)	108.7(B)	108.1(B)	161.3(D)	$+\infty$
E					108.7(D)	108.1(B)	161.3(D)	$+\infty$
F						111.9(C)	167.8(E)	176.9(E)

P=[A,B,C,D,E,F]

Algorithme de Dijkstra

P=[A,B,C,D,E,F]

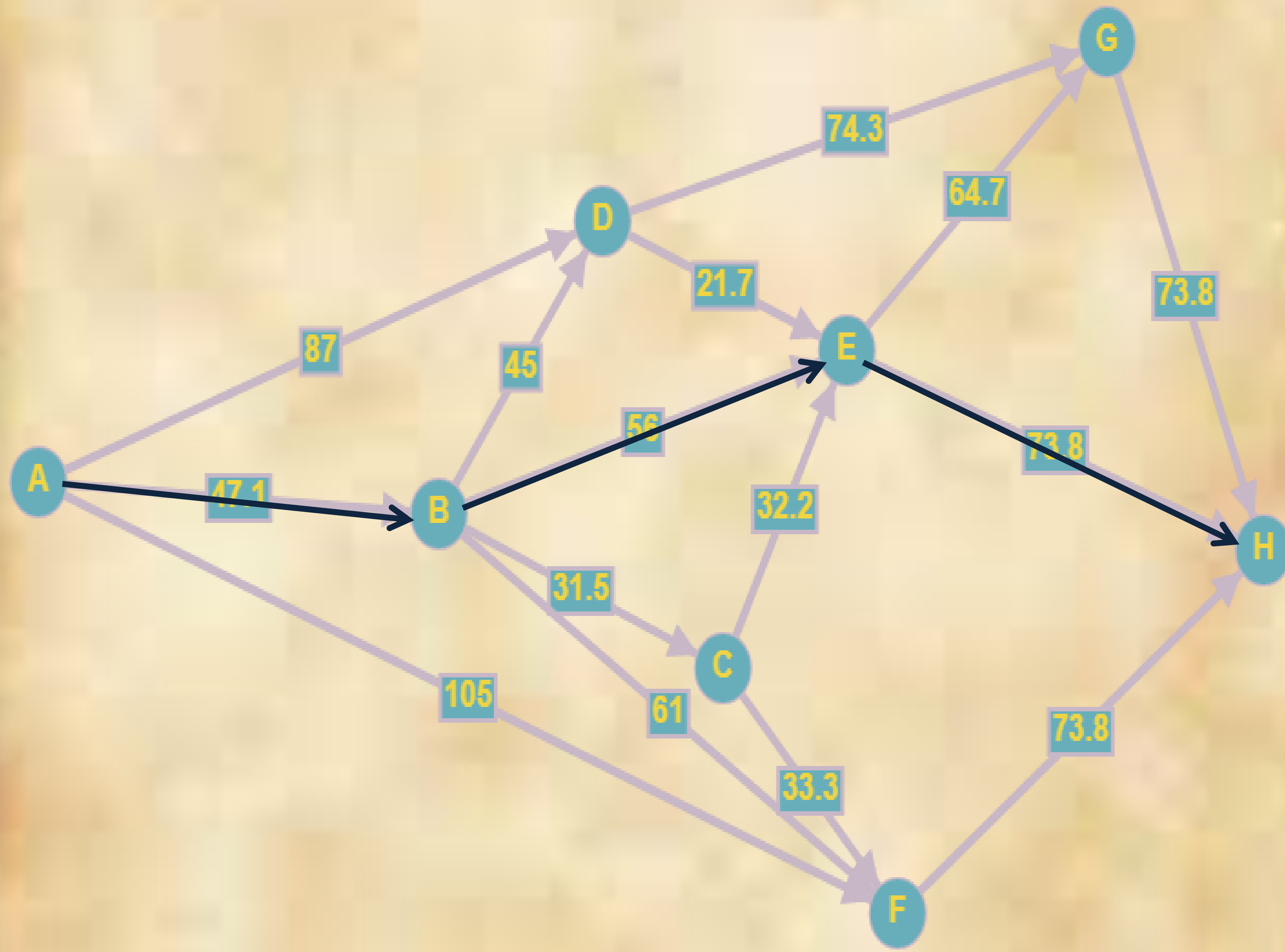
Sommet définitivement fixé	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
A	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
B		47.1(A)	$+\infty$	87(A)	$+\infty$	105(A)	$+\infty$	$+\infty$
C			78.6(B)	92.1(B)	103.1(B)	108.1(B)	$+\infty$	$+\infty$
D				92.1(B)	108.7(B)	108.1(B)	161.3(D)	$+\infty$
E					108.7(D)	108.1(B)	161.3(D)	$+\infty$
F						111.9(C)	167.8(E)	176.9(E)
G							167.8(E)	178.8(F)

P=[A,B,C,D,E,F,G]

Algorithme de Dijkstra
P=[A,B,C,D,E,F,G]

Sommet définitivement fixé	A	B	C	D	E	F	G	H
A	0	+∞	+∞	+∞	+∞	+∞	+∞	+∞
B		47.1(A)	+∞	87(A)	+∞	105(A)	+∞	+∞
C			78.6(B)	92.1(B)	103.1(B)	108.1(B)	+∞	+∞
D				92.1(B)	108.7(B)	108.1(B)	161.3(D)	+∞
E					108.7(D)	108.1(B)	161.3(D)	+∞
F						111.9(C)	167.8(E)	176.9(E)
G							167.8(E)	178.8(F)
H								235.1(G)

P=[A,B,C,D,E,F,G,H] D’où le résultat A → B → E → H


G
74.3
64.7
D
73.8
21.7
87
45
E
56
73.8
A
47.1
B
32.2
H
31.5
C
105
61
73.8
33.3
F

Calcul de complexité

La complexité, ou le coût, d'un algorithme ou d'une fonction Python est le nombre d'opérations élémentaires nécessaires à son exécution dans le pire cas.

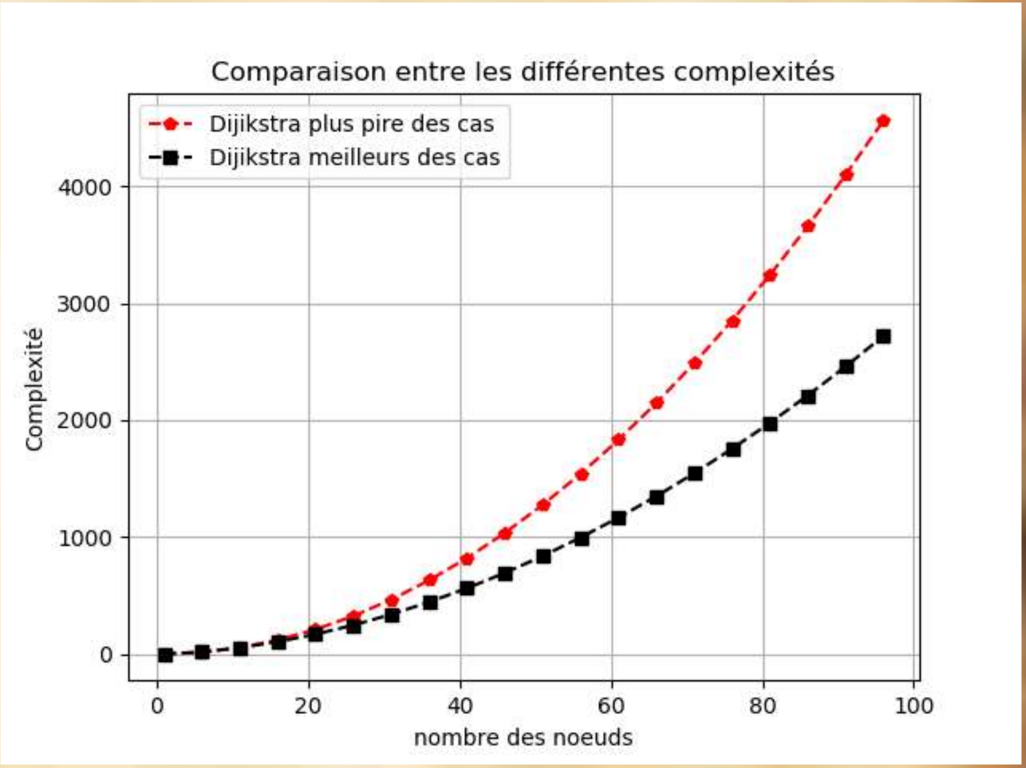
complexité temporelle

une fonction de n qui mesure le temps de calcul pour une donnée de taille n

O(m + n log(n))

n le nombre de nœuds

m le nombre d'arcs



Mais , est ce que trouver le plus court chemin maximise le flot ce qui résulte une production maximale ?

- Pour maximiser le flot on utilise l'algorithme de Ford Fulkerson

Données : un réseau G=(X,A,C)

Résultat : ϕ

ϕ= 0 — Initialisation

Répéter

chercher une chaine améliorante de s à t

si ɥ existe alors

calculer

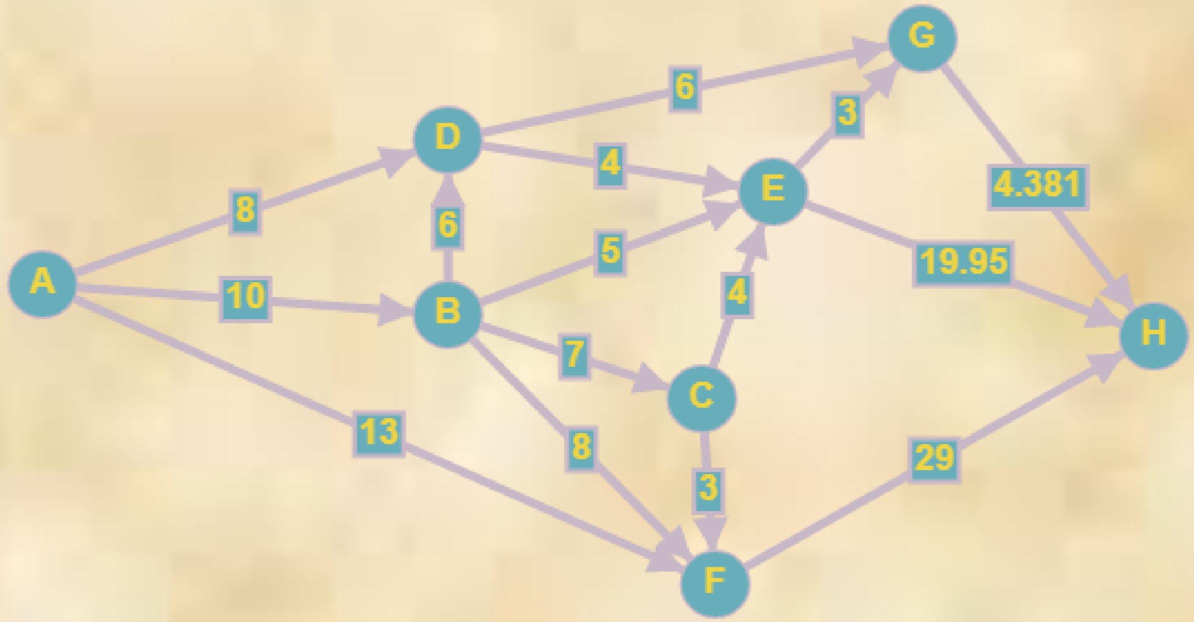
augmenter ϕ sur les arcs de ɥ+de δ

diminuer ϕ sur les arcs de ɥ-de δ

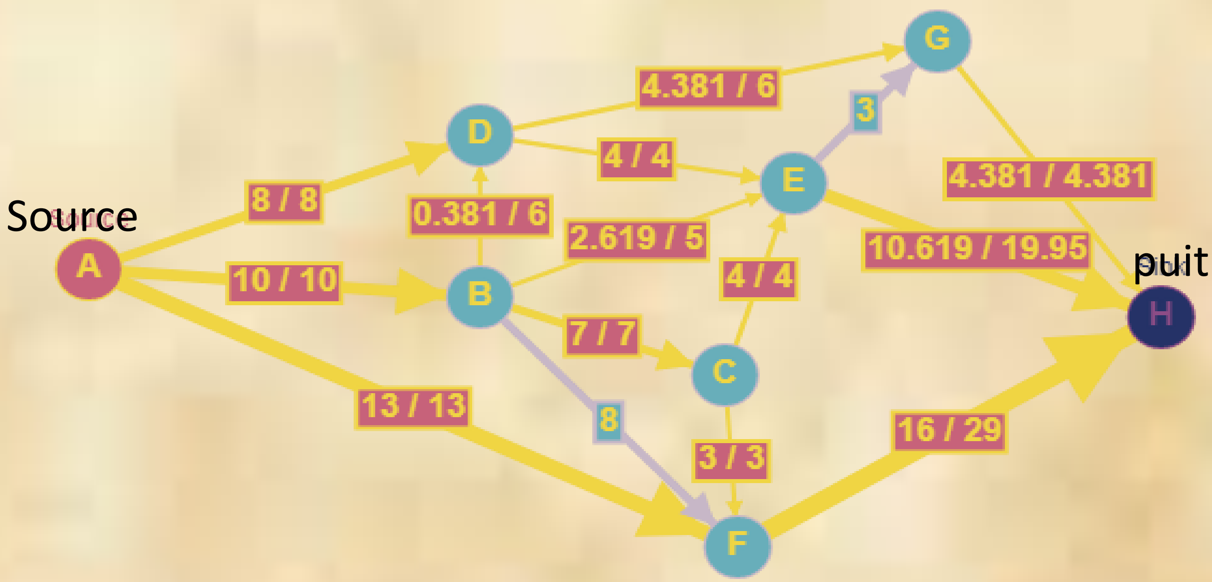
Jusqu'à il n'existe plus de chaine améliorante

Retourner ϕ

Boucle de traitements



Application de l'algorithme Ford Fulkerson



Conclusion

- Le problème de minimisation les coûts du transport de phosphate a été étudié.
- Le problème de transport optimal est compliqué nécessite d'une approche à plusieurs disciplines: mécanique de fluide, algorithmique, simulation .
- L'algorithme de Dijkstra donne le plus court chemin plus nécessairement optimal d'où utilisation de l'algorithme de Ford Fulkerson.

Annexes

Les codes python:

1. Calcul distance

```
from math import sin, cos, sqrt, atan2, radians


def distance(q,s,d,f):
    R = 6373.0
    lat1 = radians(q)
    lon1 = radians(s)
    lat2 = radians(d)
    lon2 = radians(f)

    dlon = lon2 - lon1
    dlat = lat2 - lat1

    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlon / 2)**2
    c = 2 * atan2(sqrt(a), sqrt(1 - a))

    distance = R * c

    return(distance)
```

2. Calcul Colebrook

```
import numpy as np
from scipy.optimize import fsolve

import matplotlib.pyplot as plt

def func(x):
    return (1/np.sqrt(x)) + 2*np.log10((const/3.7) + (2.51/(Re*np.sqrt(x))))

#K = 2.e-5
#D = 0.9
# const = K/D
#Re = 1e5

tab_const = np.linspace(1e-5,0.05,10)
tab_re = np.linspace(1e4,1e8,100)
tab_facteur_perte_charge = [[] for x in range(10)]
i = 0
for const in tab_const:
    for Re in tab_re:
        x0 = fsolve(func, 0.001)
        tab_facteur_perte_charge[i].append(x0)
    i = i + 1

for i in range(len(tab_facteur_perte_charge)):
    plt.plot(tab_re,tab_facteur_perte_charge[i],label=tab_const[i])
plt.xscale("log")
plt.yscale("log")
plt.xlabel("Nomre de Reynolds")
plt.ylabel("Facteur de perte des charges")
plt.title("Diagramme de Moody")
plt.legend()
plt.grid()
```

3. Réalisation d’une coupe longitudinale

```python
import urllib.request
import json
import math
import matplotlib.pyplot as plt



#START-END POINT
def conversion_degre(d,min,sec):

    return d + (min/60) + (sec/3600)

latitude1 = 36.799616666666665
longitude1 = 9.773213888888889
latitude2 = 36.550769444444444
longitude2 = 10.510544444444445

P1=[latitude1,longitude1]
P2=[latitude2,longitude2]

#NUMBER OF POINTS
s=100
interval_lat=(P2[0]-P1[0])/s #interval for latitude
interval_lon=(P2[1]-P1[1])/s #interval for longitude

#SET A NEW VARIABLE FOR START POINT
lat0=P1[0]
lon0=P1[1]

#LATITUDE AND LONGITUDE LIST
lat_list=[lat0]
lon_list=[lon0]
```


```python
#LATITUDE AND LONGITUDE LIST
lat_list=[lat0]
lon_list=[lon0]
```


```python
#GENERATING POINTS
for i in range(s):
    lat_step=lat0+interval_lat
    lon_step=lon0+interval_lon
    lon0=lon_step
    lat0=lat_step
    lat_list.append(lat_step)
    lon_list.append(lon_step)

#HAVERSINE FUNCTION
def haversine(lat1,lon1,lat2,lon2):
    lat1_rad=math.radians(lat1)
    lat2_rad=math.radians(lat2)
    lon1_rad=math.radians(lon1)
    lon2_rad=math.radians(lon2)
    delta_lat=lat2_rad-lat1_rad
    delta_lon=lon2_rad-lon1_rad
    a=math.sqrt((math.sin(delta_lat/2))**2+math.cos(lat1_rad)*math.cos(lat2_rad)
    d=2*6371000*math.asin(a)
    return d

#DISTANCE CALCULATION
d_list=[]
for j in range(len(lat_list)):
    lat_p=lat_list[j]
    lon_p=lon_list[j]
    dp=haversine(lat0,lon0,lat_p,lon_p)/1000 #km
    d_list.append(dp)
d_list_rev=d_list[::-1] #reverse list
```

```
#CONSTRUCT JSON
d_ar=[{}]*len(lat_list)
for i in range(len(lat_list)):
    d_ar[i]={"latitude":lat_list[i],"longitude":lon_list[i]}
location={"locations":d_ar}
json_data=json.dumps(location,skipkeys=int).encode('utf8')

#SEND REQUEST
url="https://api.open-elevation.com/api/v1/lookup"
response = urllib.request.Request(url,json_data,headers={'Content-Type': 'applic
fp=urllib.request.urlopen(response)

#RESPONSE PROCESSING
res_byte=fp.read()
res_str=res_byte.decode("utf8")
js_str=json.loads(res_str)
#print(js_str)
fp.close()

#GETTING ELEVATION
response_len=len(js_str['results'])
elev_list=[]
for j in range(response_len):
    elev_list.append(js_str['results'][j]['elevation'])

#BASIC STAT INFORMATION
mean_elev=round((sum(elev_list)/len(elev_list)),3)
min_elev=min(elev_list)
max_elev=max(elev_list)
distance=d_list_rev[-1]
```

```
#PLOT ELEVATION PROFILE
base_reg=0
plt.figure(figsize=(10,4))
plt.plot(d_list_rev,elev_list)
plt.plot([0,distance],[min_elev,min_elev],'--g',label='min: '+str(min_elev)+' m'
plt.plot([0,distance],[max_elev,max_elev],'--r',label='max: '+str(max_elev)+' m'
plt.plot([0,distance],[mean_elev,mean_elev],'--y',label='ave: '+str(mean_elev)+'
plt.fill_between(d_list_rev,elev_list,base_reg,alpha=0.1)
plt.text(d_list_rev[0],elev_list[0],"P1")
plt.text(d_list_rev[-1],elev_list[-1],"P2")
plt.xlabel("Distance (km)")
plt.ylabel("Altitude (m)")
plt.grid()
plt.legend(fontsize='small')
plt.show()
```

4. Ford.Fulkerson

```
def ford_fulkerson(graph, source, sink, debug=None):
    flow, path = 0, True

    while path:
        # search for path with flow reserve
        path, reserve = depth_first_search(graph, source, sink)
        flow += reserve
        # increase flow along the path
        for v, u in zip(path, path[1:]):
            if graph.has_edge(v, u):
                graph[v][u]['flow'] += reserve
            else:
                graph[u][v]['flow'] -= reserve

        # show intermediate results
        if callable(debug):
            debug(graph, path, reserve, flow)

def depth_first_search(graph, source, sink):
    undirected = graph.to_undirected()
    explored = {source}
    stack = [(source, 0, undirected[source])]

    while stack:
        v, _, neighbours = stack[-1]
        if v == sink:
            break

        # search the next neighbour
        while neighbours:
            u, e = neighbours.popitem()
            if u not in explored:
                break
        else:
            stack.pop()
            continue

        # current flow and capacity
```



```
        # current flow and capacity
```



```
        in_direction = graph.has_edge(v, u)
        capacity = e['capacity']
        flow = e['flow']
        # increase or redirect flow at the edge
        if in_direction and flow < capacity:
            stack.append((u, capacity - flow, undirected[u]))
            explored.add(u)
        elif not in_direction and flow:
            stack.append((u, flow, undirected[u]))
            explored.add(u)
    # (source, sink) path and its flow reserve
    reserve = min((f for _, f, _ in stack[1:]), default=0)
    path = [v for v, _, _ in stack]

    return path, reserve
```

```
graph = nx.DiGraph()
graph.add_nodes_from('ABCDEFGH')
graph.add_edges_from([
    ('A', 'B', {'capacity': 4, 'flow': 0}),
    ('A', 'C', {'capacity': 5, 'flow': 0}),
    ('A', 'D', {'capacity': 7, 'flow': 0}),
    ('B', 'E', {'capacity': 7, 'flow': 0}),
    ('C', 'E', {'capacity': 6, 'flow': 0}),
    ('C', 'F', {'capacity': 4, 'flow': 0}),
    ('C', 'G', {'capacity': 1, 'flow': 0}),
    ('D', 'F', {'capacity': 8, 'flow': 0}),
    ('D', 'G', {'capacity': 1, 'flow': 0}),
    ('E', 'H', {'capacity': 7, 'flow': 0}),
    ('F', 'H', {'capacity': 6, 'flow': 0}),
    ('G', 'H', {'capacity': 4, 'flow': 0}),
```