theorie_deneigeuse

July 1, 2022

[9]:
```
import numpy as np
import math
import random as rd
rd.seed(42)
```

1 Passage Deneigeuse :

hyptohese:

On modelise les rues via un graphe Le graphe sera orienté (sens de circulation) Chercher à parcourir toute les rues revient à chercher un chemin eulérien dans un graphe orienté Le deneigeuse devra revenir à sont point de départ -> cycle eulérien le graphe peut ne pas avoir de cycle eulérien, à nous de l'adapter à cet objectif Premier problème (chinese postman oriented problem):

Un cycle eulérien est soumis à l'equilibre des sommets Idée:

lié chaque noeud avec un déséquilibre négatif avec un noeud de déséquilibre négatif.

premiere implementation : choisir la valeur la plus petite et continuer jusqu'à rendre le graphe eulérien

optimisation des couts: utilisation de bellman-ford

[36]:
```
def ExistChemin(matriceAdj, u, v):
    n = len(matriceAdj)
    file = []
    visites = [False] * n
    file.append(u)
    while file:
        courant = file.pop(0)
        visites[courant] = True
        for i in range(n):
            if matriceAdj[courant][i] > 0 and i == v:
                return True
            elif matriceAdj[courant][i] > 0 and not visites[i]:
                file.append(i)
                visites[i] = True

    return False
```

```
class Graph:
    def __init__(self,nb_vertices):
        self.delta = [0] * nb_vertices
        self.defined = np.zeros((nb_vertices,nb_vertices)).astype(bool)
        self.c = np.zeros((nb_vertices,nb_vertices))
        self.arcs = np.zeros((nb_vertices,nb_vertices))
        self.N = nb_vertices
        self.neg = []
        self.pos = []

    def addArc(self, u, v, cost):
        if (cost < 0):
            raise "Cost must be positive"
        self.c[u][v] = cost if self.c[u][v] == 0 else min(cost,self.c[u][v])
        self.defined[u][v] = True
        self.arcs[u][v] += 1
        self.delta[u] += 1
        self.delta[v] -= 1

    def connected(self):
      for i in range(self.N):
          for j in range(self.N):
              if (i != j) and ExistChemin(self.arcs, j, i) == False:
                  return False
      return True

    def is_eulerian_directed(self):
      self.findUnbalanced()
      return self.connected() and len(self.neg) == 0 and len(self.pos) == 0

    def findUnbalanced(self):
        nn = 0
        np = 0
        for i in range(self.N):
            if self.delta[i] < 0:
                nn += 1
            elif self.delta[i] > 0:
                np += 1

        self.neg = [0] * nn
        self.pos = [0] * np
        np = nn = 0
        for i in range(self.N):
            if self.delta[i] < 0:
                self.neg[nn] = i
                nn += 1
```

```
            elif self.delta[i] > 0:
                self.pos[np] = i
                np += 1

    def BellmanFord(self,src):
        n = self.N
        dist = [math.inf] * n
        dist[src] = 0
        edges = []
        for i in range(self.N):
            for j in range(self.N):
                if (self.defined[i][j]):
                    edges.append((i,j,self.c[i][j]))
        for k in range(n):
            for (s, d, w) in edges:
                dist[d] = min(dist[d], dist[s] + w)
        return dist

    def balas_hammer(self):
        nead_in = np.zeros(len(self.pos))
        nead_out = np.zeros(len(self.neg))
        for i in range(len(self.neg)):
            nead_out[i] = -self.delta[self.neg[i]]
        for i in range(len(self.pos)):
            nead_in[i] = self.delta[self.pos[i]]
        if (nead_in.sum() != nead_out.sum()):
            raise "Can't balance graph with ballas hammer"
        cost = np.zeros((len(self.neg),len(self.pos)))
        for i in range(len(self.pos)):
            ballas = self.BellmanFord(self.pos[i])
            for j in range(len(self.neg)):
                cost[i][j] = ballas[self.neg[j]]
        while (nead_in.sum() != 0):
            delta_line = np.zeros(len(self.pos))
            for i in range(len(self.pos)):
                delta_line[i] = cost[i].max()-cost[i].min()
            delta_col = np.zeros(len(self.neg))
            for i in range(len(self.pos)):
                delta_col[i] = cost[:,i].max()-cost[:,i].min()
            max_line = delta_line.argmax()
            max_col = delta_col.argmax()
            if (delta_line[max_line] > delta_col[max_col]):
                mini = cost[max_line].argmin()
                self.addArc(self.neg[mini], self.pos[max_line], cost[max_line][mini])
                print("Add arc:", self.neg[mini], self.pos[max_line], cost[max_line][mini])
```

```
                nead_in[max_line] -= 1
                nead_out[mini] -= 1
                if (nead_in[max_line] == 0):
                    nead_in = np.delete(nead_in, max_line)
                    self.pos = np.delete(self.pos, max_line)
                    cost = np.delete(cost, max_line, axis=0)
                if (nead_out[mini] == 0):
                    nead_out = np.delete(nead_out, mini)
                    self.neg = np.delete(self.neg, mini)
                    cost = np.delete(cost, mini, axis=1)
            else:
                mini = cost[:,max_col].argmin()
                self.addArc(self.neg[max_col], self.pos[mini], cost[mini][max_col])
                print("Add arc:", self.neg[max_col], self.pos[mini], cost[mini][max_col])
                nead_out[max_col] -= 1
                nead_in[mini] -= 1
                if (nead_out[max_col] == 0):
                    nead_out = np.delete(nead_out, max_col)
                    self.pos = np.delete(self.pos, max_col)
                    cost = np.delete(cost, max_col, axis=1)
                if (nead_in[mini] == 0):
                    nead_in = np.delete(nead_in, mini)
                    self.neg = np.delete(self.neg, mini)
                    cost = np.delete(cost, mini, axis=0)

    def edges(self):
        edges = []
        arcs = self.arcs.copy()
        for i in range(self.N):
            for j in range(self.N):
                while (arcs[i][j] > 0):
                    edges.append((i,j))
                    arcs[i][j] -= 1
        return edges

    def solve(self):
        if (not self.connected()):
            raise "Graph is not connected"
        self.findUnbalanced()
        if (not self.is_eulerian_directed()):
            print("Graph is not eulerian")
            print("Balancing with ballas hammer")
            print("Edges before edge balancing:", self.edges())
            self.balas_hammer()
            print("Edges after edge balancing:", self.edges())
```

```
            print("is_eulerian: ", G.is_eulerian_directed())
        return self.eularian_cycle()


    def eularian_cycle(self):
        edges = self.edges()
        if len(edges) == 0:
            return []
        cycle = [edges[0][0]]
        while True:
            rest = []
            for (a, b) in edges:
                if cycle[-1] == a:
                    cycle.append(b)
                else:
                    rest.append((a,b))
            if not rest:
                assert cycle[0] == cycle[-1]
                return cycle[0:-1]
            edges = rest
            if cycle[0] == cycle[-1]:
                for (a, b) in edges:
                    if a in cycle:
                        idx = cycle.index(a)
                        cycle = cycle[idx:-1] + cycle[0:idx+1]
                        break
```

[37]:

```
G = Graph(4)
G.addArc(0, 1, 1)
G.addArc(1, 2, 1)
G.addArc(2, 3, 1)
G.addArc(3, 0, 1)
G.addArc(1, 3, 1)
G.addArc(0, 2, 1)
print("is_eulerian: ", G.is_eulerian_directed())
print("eulerian cycle: ",G.solve())
```

```
is_eulerian:  False
Graph is not eulerian
Balancing with ballas hammer
Edges before edge balancing: [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 0)]
Add arc: 3 1 1.0
Add arc: 2 0 1.0
Edges after edge balancing: [(0, 1), (0, 2), (1, 2), (1, 3), (2, 0), (2, 3), (3,
0), (3, 1)]
is_eulerian:  True
eulerian cycle:  [1, 2, 0, 2, 3, 0, 1, 3]
```