TD-TP sur les Flots

Ce "TD-TP" est, comme son nom l'indique, constitué une partie TD à faire sur papier & d'une partie TP à faire sur machine.

Il n'est pas nécessaire de faire tout le TD avant d'attaquer le TP, néanmoins, avant de programmer un algorithme, il faut l'écrire à la main.

On considère un graphe orienté $G=(V,A)$ tel qu'à chaque arc (arête orientée) u soit associé un nombre positif c_u, la **capacité** de u. Un **flot (réalisable)** est une suite $\Phi = (\phi_u, u\in A)$ telle que :

- Φ est compatible avec les capacités : $\forall u \in A, 0\leq \phi_u \leq c_u$
- Φ respecte la loi de Kirchoff (ou loi des nœuds) : $\forall v \in V, \sum_{u \in \Gamma^+(v)}{\phi_u} = \sum_{u \in \Gamma^-(v)}{\phi_u}$
  $\Gamma^+(v)$ est l'ensemble des arcs *entrant* en v (*i.e.* qui vont d'un sommet v' vers v) & $\Gamma^-(v)$ est l'ensemble des arcs *sortant* de v.

Étant donnés deux sommets s & p (la **source** & le **puit**), le problème du **flot maximum** est de faire passer le plus grand flot de s à p.

Pour se faire une idée un peu plus concrète que ce qu'est un flot, on peut imaginer chaque arc comme un tuyau à sens unique (en fait, cette restriction n'est pas nécessaire — cf Exercice 3) ayant un débit maximum (sa capacité), le problème étant de faire passer le plus de liquide possible de la *source* au *puit*.

L'intérêt des flots est qu'ils permettent de modéliser un grand nombre de problèmes *concrets* (*i.e.* du monde *réel* — & pas seulement pour les plombiers), par exemple des problèmes d'affectation de tâches (connaissant les capacités/appétences de différentes personnes, comment répartir entre elles des tâches à accomplir) ou de logistique (organisation de réseaux de transport). & ce d'autant plus qu'il en existe un grand nombre de variantes (flots *compatibles*, *à coût minimum*, *avec multiplicateurs*, ...)

Partie TD

L'exercice 1 est dédié à l'écriture d'un algorithme de flot maximum, les suivants à la modélisation de problèmes *concrets* par un problème de flot.

1. Un algorithme de flot

Tout d'abord (& pas seulement pour cet algorithme), on rajoute au graphe un **arc de retour** (p,s).
Quel est l'intérêt de cet arc ? Réponse
Quelle est sa capacité ? Indication

Étant donné un flot Φ sur un graphe $G=(V,A)$, on considère le **graphe d'écart** G_Φ, avec les mêmes sommets que G &, pour chaque arc $u=(x,y)$ de A, deux arcs $u^+ = (x,y)$, de capacité $c_u - \phi_u$ & $u^- = (y,x)$ de capacité ϕ_u.

Les capacités des arcs u^+ & u^- sont appelées **capacités résiduelles**, elles correspondent aux variations (augmentations ou diminutions) maximales de flux que l'on peux faire sur l'arc u : la capacité de u^+ à l'augmentation & celle de u^- à la diminution.
Remarquer que diminuer le flux sur l'arc $u=(x,y)$ d'une quantité q revient à ajouter un flux de valeur q dans le sens (y,x).

En fait, dans le graphe d'écart, on ne met QUE les arcs de capacité résiduelle strictement positive.

À quoi correspond un chemin de de s à p dans G_Φ ? Indication

En déduire un algorithme de flux maximum. Réponse

2. Mariages

Une agence matrimoniale a, comme clients, n hommes & p femmes. Parmi les $n\times p$ couples homme-femme[1)] possibles, certains sont *compatibles*. Le problème est de créer le plus possible de couples (compatibles).

On modélisera ce problème par un flot maximum dans un graphe que l'on déterminera. Indication

3. Graphes non orientés

Montrez que l'algorithme des graphes d'écart peut s'adapter aux graphes non orientés. Indication

4. La bataille de la Marne

On a un ensemble S de villes & des routes reliant certaines villes entre elles (il peut exister plusieurs routes entre deux villes). Chaque ville i est caractérisée par un nombre p_i de places de parking, & chaque route j par une longueur l_j (le temps pour aller d'une extrémité à l'autre) & une capacité c_j (nombre de véhicules pouvant l'emprunter par unité de temps).

Au temps $t=0$, un certain de nombre de véhicules sont stationnés dans différentes villes ; il faut qu'au temps $t=K$, le plus possible de véhicules soient arrivés à une ville donnée (la Marne). Il est possible que des véhicules arrivent avant cette date butoir, mais après la date K, c'est trop tard.

On modélisera ce problème par un flot maximum dans un graphe que l'on déterminera. Indication

5. Des extensions

On peut supposer que chaque arc u a une capacité minimum c_u & une capacité maximum c^u (on doit avoir $c_u\leq \phi_u \leq c^u$). Le problème est alors de trouver un flux **compatible** (qui n'existe pas forcément).

On peut aussi supposer que chaque arc prélève un "péage", proportionnel à la quantité de flux qui y passe. On a alors le problème du **flux compatible à coût minimum**.

Montrez que la problème du plus court chemin (entre deux sommets x) & y dans un graphe valué peut se modéliser par un problème de flux compatible à coût minimum dans un graphe que l'on déterminera. Indication

6. Un théorème

Si toutes les capacités sont des entiers, que peut-on dire des flots maximums? Indication

Partie TP

On va maintenant implémenter (presque tout) ce qui a été vu dans la partie TD.

Le fait de tester toutes vos créations n'est plus explicitement demandé car cela doit **encore & toujours** être fait.

1. Implémentation des graphes

Les graphes seront représentés "mathématiquement" par des listes d'adjacence, & plus précisément/concrètement par des dictionnaires de dictionnaires de dictionnaires. Par exemple :

```
CAPACITE = "capacite"
FLUX = "flux"

the_graph = {1: {2: {CAPACITE: 5, FLUX: 2}, 3:{CAPACITE: 6,
FLUX: 1}},
             2: {3: {CAPACITE: 7, FLUX: 1}, 4: {CAPACITE: 8,
FLUX: 1}},
            3: {5: {CAPACITE: 4, FLUX: 2}},
            4: {6: {CAPACITE: 3, FLUX: 2}},
             5: {4: {CAPACITE: 6, FLUX: 1}, 6: {CAPACITE: 8,
```

```
FLUX: 1}},
             6: {}
             }
```

Dessinez le graphe ci-dessus. Indication

2. L'algorithme des graphes d'écart

Implémentez l'Algorithme des graphes d'écart. On pourra utiliser les fonctions suivantes Indication :

```
import math

INFINI = math.inf
CAPACITE = "capacite"
FLUX = "flux"


def mise_a_zero_flot(graph):
    for sommet in graph:
        for voisin in graph[sommet]:
            graph[sommet][voisin][FLUX] = 0


def construction_graphe_ecart(graphe_antisymetrique):
    graphe_ecart = {}
    for sommet in graphe_antisymetrique:
        graphe_ecart[sommet] = {}
    for sommet in graphe_antisymetrique:
        for voisin in graphe_antisymetrique[sommet]:
            capacite = graphe_antisymetrique[sommet][voisin][CAPACITE]
            flux = graphe_antisymetrique[sommet][voisin][FLUX]
            if capacite > flux:
                graphe_ecart[sommet][voisin] = capacite - flux
            if flux > 0:
                graphe_ecart[voisin][sommet] = flux
    return graphe_ecart


def construction_chemin_via_dfs(graphe_ecart, source, puits):
    sommets_marques = {}
    antecedents = {}
    pile = [source]
    for sommet in graphe_ecart:
        antecedents[sommet] = None
    while pile != []:
        sommet = pile.pop()
        if sommet not in sommets_marques:
            sommets_marques[sommet] = True
```

```
            for voisin in graphe_ecart[sommet]:
                if voisin not in sommets_marques:
                    antecedents[voisin] = sommet
                    pile.append(voisin)
        if sommet == puits:
            return construction_effective_chemin(sommet, antecedents)


def construction_effective_chemin(sommet, antecedents):
    chemin = []
    while sommet is not None:
        chemin.append(sommet)
        sommet = antecedents[sommet]
    chemin.reverse()
    return  chemin


def min_sur_chemin(chemin, graphe_ecart):
    minimum = INFINI
    for i in range(len(chemin) - 1):
        courant = chemin[i]
        suivant = chemin[i + 1]
        if graphe_ecart[courant][suivant] < minimum:
            minimum = graphe_ecart[courant][suivant]
    return minimum


def ajout_flot(valeur, chemin, graphe_antisymetrique):
    for i in range(len(chemin) - 1):
        courant = chemin[i]
        suivant = chemin[i + 1]
        if suivant in graphe_antisymetrique[courant]:
            graphe_antisymetrique[courant][suivant][FLUX] += valeur
        else:
            graphe_antisymetrique[suivant][courant][FLUX] -= valeur
```

3. Le problème du mariage

Résoudre le problème du mariage avec les données suivantes :

```
CLEOPATRE = "Cleopatre"
IPHIGENIE = "Iphigenie"
JULIETTE = "Juliette"
FANNY = "Fanny"
CHIMENE = "Chimene"

ACHILLE = "Achille"
CESAR = "Cesar"
RODRIGUE = "Rodrigue"
ROMEO = "Romeo"
```

```
MARIUS = "Marius"

LES_COUPLES = [(CLEOPATRE, ACHILLE), (CLEOPATRE, CESAR), (CLEOPATRE, ROMEO),
               (IPHIGENIE, ACHILLE), (JULIETTE, CESAR), (JULIETTE,
RODRIGUE), (JULIETTE, ROMEO),
               (FANNY, CESAR), (FANNY, MARIUS), (CHIMENE, RODRIGUE),
(CHIMENE, ROMEO)]
```

La difficulté (puisque l'algorithme de flots est (normalement) déjà écrit) est la construction du graphe à partir de la liste LES_COUPLES. Ceci doit bien sûr être fait par un programme & non pas à la main.

4. La bataille de la Marne

Résoudre le problème de "la bataille de la Marne", en supposant que,

- Au temps 0, on a autant de taxis que nécessaire dans la ville 14 ("Paris").
- Au temps 50, il faut qu'il y en ait le plus possible dans la ville 0 ("La Marne").
- Chaque ville (intermédiaire) a 10 places de parking (Paris en a autant que nécessaire).
- Les routes sont données dans le tableau suivant :

```
LES_ROUTES = [{'depart': 0, 'capacite': 2, 'arrivee': 1, 'longueur': 5},
{'depart': 0, 'capacite': 3, 'arrivee': 2, 'longueur': 3},
{'depart': 0, 'capacite': 6, 'arrivee': 1, 'longueur': 6},
{'depart': 1, 'capacite': 3, 'arrivee': 2, 'longueur': 6},
{'depart': 1, 'capacite': 7, 'arrivee': 4, 'longueur': 7},
{'depart': 1, 'capacite': 6, 'arrivee': 2, 'longueur': 7},
{'depart': 2, 'capacite': 7, 'arrivee': 4, 'longueur': 4},
{'depart': 2, 'capacite': 6, 'arrivee': 4, 'longueur': 8},
{'depart': 2, 'capacite': 2, 'arrivee': 5, 'longueur': 5},
{'depart': 3, 'capacite': 7, 'arrivee': 5, 'longueur': 5},
{'depart': 3, 'capacite': 3, 'arrivee': 5, 'longueur': 4},
{'depart': 3, 'capacite': 7, 'arrivee': 6, 'longueur': 4},
{'depart': 4, 'capacite': 7, 'arrivee': 5, 'longueur': 8},
{'depart': 4, 'capacite': 8, 'arrivee': 7, 'longueur': 3},
{'depart': 4, 'capacite': 4, 'arrivee': 7, 'longueur': 3},
{'depart': 5, 'capacite': 8, 'arrivee': 6, 'longueur': 8},
{'depart': 5, 'capacite': 4, 'arrivee': 7, 'longueur': 7},
{'depart': 5, 'capacite': 4, 'arrivee': 6, 'longueur': 5},
{'depart': 6, 'capacite': 5, 'arrivee': 9, 'longueur': 3},
{'depart': 6, 'capacite': 5, 'arrivee': 7, 'longueur': 3},
{'depart': 6, 'capacite': 7, 'arrivee': 8, 'longueur': 3},
{'depart': 7, 'capacite': 2, 'arrivee': 8, 'longueur': 8},
{'depart': 7, 'capacite': 2, 'arrivee': 10, 'longueur': 5},
{'depart': 7, 'capacite': 7, 'arrivee': 10, 'longueur': 4},
{'depart': 8, 'capacite': 3, 'arrivee': 11, 'longueur': 4},
{'depart': 8, 'capacite': 5, 'arrivee': 9, 'longueur': 3},
{'depart': 8, 'capacite': 8, 'arrivee': 10, 'longueur': 3},
```

```
{'depart': 9, 'capacite': 1, 'arrivee': 11, 'longueur': 8},
{'depart': 9, 'capacite': 4, 'arrivee': 12, 'longueur': 3},
{'depart': 9, 'capacite': 1, 'arrivee': 11, 'longueur': 7},
{'depart': 10, 'capacite': 5, 'arrivee': 12, 'longueur': 7},
{'depart': 10, 'capacite': 3, 'arrivee': 13, 'longueur': 3},
{'depart': 10, 'capacite': 7, 'arrivee': 11, 'longueur': 4},
{'depart': 11, 'capacite': 4, 'arrivee': 14, 'longueur': 8},
{'depart': 11, 'capacite': 2, 'arrivee': 12, 'longueur': 3},
{'depart': 11, 'capacite': 6, 'arrivee': 14, 'longueur': 6},
{'depart': 12, 'capacite': 5, 'arrivee': 13, 'longueur': 6},
{'depart': 12, 'capacite': 2, 'arrivee': 14, 'longueur': 7},
{'depart': 12, 'capacite': 2, 'arrivee': 14, 'longueur': 6},
{'depart': 13, 'capacite': 2, 'arrivee': 14, 'longueur': 5},
{'depart': 13, 'capacite': 2, 'arrivee': 14, 'longueur': 3},
{'depart': 13, 'capacite': 8, 'arrivee': 14, 'longueur': 6}]
```

Les routes sont à double sens (il ne faut pas se fier aux appellations "départ" & "arrivée").

La difficulté est, là encore, la construction du graphe.

On pourra commencer par une instance plus petite, par exemple en ne considérant que les villes jusqu'à 9, avec un borne max pour le temps de 15, c'est à dire avec :

```
LES_ROUTES = [{'depart': 0, 'capacite': 2, 'arrivee': 1, 'longueur': 5},
{'depart': 0, 'capacite': 3, 'arrivee': 2, 'longueur': 3},
{'depart': 0, 'capacite': 6, 'arrivee': 1, 'longueur': 6},
{'depart': 1, 'capacite': 3, 'arrivee': 2, 'longueur': 6},
{'depart': 1, 'capacite': 7, 'arrivee': 4, 'longueur': 7},
{'depart': 1, 'capacite': 6, 'arrivee': 2, 'longueur': 7},
{'depart': 2, 'capacite': 7, 'arrivee': 4, 'longueur': 4},
{'depart': 2, 'capacite': 6, 'arrivee': 4, 'longueur': 8},
{'depart': 2, 'capacite': 2, 'arrivee': 5, 'longueur': 5},
{'depart': 3, 'capacite': 7, 'arrivee': 5, 'longueur': 5},
{'depart': 3, 'capacite': 3, 'arrivee': 5, 'longueur': 4},
{'depart': 3, 'capacite': 7, 'arrivee': 6, 'longueur': 4},
{'depart': 4, 'capacite': 7, 'arrivee': 5, 'longueur': 8},
{'depart': 4, 'capacite': 8, 'arrivee': 7, 'longueur': 3},
{'depart': 4, 'capacite': 4, 'arrivee': 7, 'longueur': 3},
{'depart': 5, 'capacite': 8, 'arrivee': 6, 'longueur': 8},
{'depart': 5, 'capacite': 4, 'arrivee': 7, 'longueur': 7},
{'depart': 5, 'capacite': 4, 'arrivee': 6, 'longueur': 5},
{'depart': 6, 'capacite': 5, 'arrivee': 9, 'longueur': 3},
{'depart': 6, 'capacite': 5, 'arrivee': 7, 'longueur': 3},
{'depart': 6, 'capacite': 7, 'arrivee': 8, 'longueur': 3},
{'depart': 7, 'capacite': 2, 'arrivee': 8, 'longueur': 8},
{'depart': 7, 'capacite': 2, 'arrivee': 10, 'longueur': 5},
{'depart': 7, 'capacite': 7, 'arrivee': 10, 'longueur': 4},
```

```
{'depart': 8, 'capacite': 3, 'arrivee': 11, 'longueur': 4},
{'depart': 8, 'capacite': 5, 'arrivee': 9, 'longueur': 3}]
```

voire même plus petit.

1)

Il est tout à fait possible de considérer d'autres couples, mais le problème est plus compliqué.