Maintenance sur un réseau électrique

MOD 4.4 - Recherche Opérationnelle

October 10, 2022

Abstract

Le sujet de ce BE est inspité du challenge de la ROADEF de l'année 2020 proposé par RTE (gestionnaire de distribution du réseau électrique en France). La ROADEF est la Société française de recherche opérationnelle et d'aide à la décision. Elle organise tous les deux ans un concours de programmation proposée par un industriel pour résoudre un problème "réel" à destination des chercheurs et doctorants travaillant sur les problème de Recherche Opérationnelle.

1 Travail à accomplir

Le travail attendu est à coder en Python et le sujet est pensé en ce sens.

Il faudra rendre au plus 3 semaines après le BE:

- Un rapport **au format PDF** avec la réponse aux questions théoriques, des logs de vos différentes fonctions.
- Une archive avec **tous vos fichiers sources** !

Les questions avec des (*) sont plus difficiles. Le but du BE est de traiter -à minima- les questions sans (*). Si vous traiter correctement des questions avec une (*) vous obtiendrez des points supplémentaires.

Les documents sont à envoyer par e-mail à l'adresse **nicolas.bousquet@univ-lyon1.fr**.

Remarque: Le sujet propose des noms de fonctions. Si vous ne respectez ces noms, un malus sera appliqué.

2 Introduction

RTE est le Gestionnaire du Réseau de Transport d'Electricité français. Autrement dit, RTE gère les opérations sur le réseau français d'électricité. De nombreuses opérations, soit pour de la maintenance soit pour réparer des lignes endommagées, sont programmées et réalisées chaque jour sur le réseau. Pourtant ces règles doivent être appliquées avec soin puisqu'elle nécéssite de "couper" le courant sur certaines lignes. Ce "courant" doit donc être redirigé vers d'autres lignes afin de ne pas couper le courant des utilisateurs finaux.

De plus, pour ces interventions, certaines ressources sont nécessaires. Des ressources humaines mais aussi materielles qui rajoutent des contraintes supplémentaires puisqu'elles empêchent certaines opérations d'être faite simultanement (si, par exemple, deux opérations sur des lignes aériennes doivent être effectuée, il faut être sur d'avoir deux camions avec une échelle pour pouvoir les faire ou alors il faut les programmer à des dates différentes).

Mais il faut aussi que certaines interventions soient effectuée à des dates différentes puisqu'elles nécessitent d'opérer sur des lignes trop proches géographiquement ce qui peut créer des coupures

de courant locales sur le réseau. Les véritables contraintes sont en fait encore plus difficiles à évaluer.

3 Modélisation (simplifiée) du problème proposé par RTE

3.1 Ensembles sans conflits

Les données du problème sont les suivantes:

- Un ensemble $\mathcal{T} = T_1, \ldots, T_r$ de tâches à effectuer. Pour simplifier, on supposera lors de ce BE que chaque tâche prend un temps unitaire à être effectuée.
- Un ensemble de ℓ ressources $R_1, \ldots, R_\ell$ telles que, pour chaque ressource, on a une quantité limitée r_i de cette ressource. Autrement dit, à chaque étape, on ne peut utiliser que r_i unités de ressource R_i.
- Un ensemble de r vecteurs $s_1, \ldots, s_r$ de taille ℓ tel que la j-eme coordonnée du vecteur s_i nous dit combien d'unités de la ressource j sont nécessaires pour effectuer la tâche T_i. On denotera par $s_i(j)$ la j-ieme coordonnée de s_i.
- Un graphe G de conflits qui donne un ensemble de tâches qui ne peuvent pas être effectuées simultanement deux à deux. Autrement dit, les sommets de G sont les tâches et il y a une arête entre deux sommets si les tâches correspondantes ne peuvent pas être effectuées simultanément.
  On notera V l'ensemble de sommets de G et E l'ensemble d'arêtes de G.

Un sous-ensemble $\mathcal{T}'$ de tâches $\mathcal{T}$ est *sans conflit* si:

- Il n'y a pas d'arêtes de G entre les paires de tâches dans $\mathcal{T}'$.
- Pour chaque ressource R_j, les tâches dans $\mathcal{T}'$ doivent, au total, utiliser au plus r_j unités de ressource R_j.

Le but du problème est de trouver un ensemble sans conflit de taille maximum. Autrement dit, de faire autant de tâches que possible.

Question 1.
Modéliser, sous la forme d'équation, les contraintes qui doivent être satisfaites par un ensemble $\mathcal{T}'$ pour être sans conflit.

Question 2.
En déduire un Programme Linéaire en Nombre Entiers (PLNE) dont la solution optimale consiste à trouver un ensemble $\mathcal{T}'$ de $\mathcal{T}$ sans conflit de taille maximum.

3.2 Simplification du problème

Dans la suite on va encore simplifier le problème et oublier les ressources. Autrement dit, on va considérer le problème suivant:

On se donne un graphe G. Let but est de trouver un sous ensemble de sommets $X \subseteq V$ de taille maximum tels que les sommets de X sont deux à deux non adjacents. (Autrement dit, on veut trouver un ensemble sans contraintes de taille maximum). Un tel ensemble est appelé un *ensemble indépendant de G*.

Question 3.
Simplifier le programme linéaire de la Question 2 afin qu'il trouve un ensemble indépendant de taille maximum.

4 Implémentation en Python

Soit $G = (V, E)$ un graphe où V est l'ensemble de sommets et E le nombre d'arêtes.

La *matrice d'adjacence* A d'un graphe est une matrice à $|V|$ lignes et $|V|$ colonnes telle que si (i, j) est une arête alors $A(i, j) = A(j, i) = 1$. Sinon $A(i, j) = 0$.

En général les matrices d'adjacence de graphes sont creuses (le nombre d'arêtes dans les "vrais" graphes est souvent linéaire). Une représentation par liste plutôt qu'une représentation matricielle permet donc de gagner de l'espace.

Une matrice d'incidence d'un graphe G est une liste de $|V|$ listes telle que la liste du sommet v est la liste des voisins de v. Autrement dit, la liste associée au sommet V, est la liste des sommets w tels que vw est une arête.

Question 4.a. Écrire une fonction Adj_vers_Inc(G) qui étant donné une matrice d'adjacence renvoie la matrice d'incidence.

Question 4.b. Écrire une fonction Inc_vers_Adj(G) qui étant donné la matrice d'incidence d'un graphe renvoie sa matrice d'adjacence.

Question 5.
Le chemin P_n a n sommets est le chemin tel que, pour tout $i \leq n - 1$, les sommets i et $(i + 1)$ sont adjacents. Le cycle C_n est le graphe P_n plus l'arête $n1$. Un clique K_n est le graphe à n sommets où tous les sommets sont deux à deux reliés.

Question 5.a.
Ecrire des fonctions Chemin(n), Cycle(n) et Clique(n) qui renvoie la matrice d'incidence de P_n, C_n et K_n respectivement.

Question 5.b. Que renvoie Inc_vers_Adj(G) pour P_5, C_5, K_5.

4.1 Ouverture de fichiers extérieurs

Certains des graphes sur lesquels on aimerait tester nos algorithmes sont disponible à la page web suivante: `https://perso.liris.cnrs.fr/nbousquet/ECL_RO/be_RTE.py`. Le fichier donne un graphe sous la forme d'un *stream* (format .gr). La première ligne donne le nombre de sommets et d'arêtes du graphe sous la forme suivante:

p cep NombreDeSommets NombreDAretes

Les sommets sont numérotés de 1 à NombreDeSommets. Ensuite chaque ligne est de la forme "Entier1 Entier2" et donne l'existence d'une arête entre les sommets étiquetés Entier1 et Entier2.

Le fichier be_RTE.py contient trois fonctions:

- $nom\_fichier(i)$ qui étant donné un entier i va renvoyer l'adresse du i-ème fichier de type graphe.
- $nombre\_sommets(i)$ qui étant donné un entier i va renvoyer le nombre de sommets du i-ème fichier

Attention aux problemes d'indices ! Les listes en Python sont indexées à partir de 0 alors que les sommets des graphes sont étiquetés à partir de 1 ! Afin d'éviter les problèmes d'indices, les sommets sont renommés dans $listeadj(i)$. Autrement dit si on lit l'arête $A\ B$ on crée en fait l'arête entre les sommets $A - 1$ et $B - 1$.

4.2 Algorithme glouton

Une façon de trouver un ensemble sans conflit est de développer une heuristique pour le faire. Nous proposons l'heuristique suivante:

Procedure 1 Algorithme glouton pour l'Ensemble Sans Conflit

Input: $G = (V, E)$ un graphe.
Output: X un ensemble sans conflit de G.
$X = \emptyset$
$G' = G$
while G' contient un sommet **do**
Rajouter un sommet v de degré minimum dans X.
$G' \leftarrow G' \setminus N(v)$ (le graphe G' où v et tous les voisins de v sont supprimés; on supprime également toutes les arêtes qui leur sont incidentes. Autrement dit le degré d'un sommet restant w est diminué de ℓ si w avait ℓ voisins dans $v \cup N(v)$.)
end while
Renvoyer X

Le *degré* d'un sommet v est le nombre d'arêtes qui contiennent ce sommet. Autrement dit c'est le nombre de sommets qui sont adjacents à v (que l'on appelle les *voisins* de v).

Question 6.a.
Ecrire une fonction Matricedegre(G) qui renvoie la liste des degrés des sommets du graphe étant donné un graphe G donné sous forme de liste d'incidence.

Question 6.b.
Ecrire une fonction degre_minimum(L) qui renvoie le degré minimum du graphe où L est la liste des degré du graphe.

Question 6.c
Ecrire une fonction MiseAJourDegre(G,L,v) qui étant donné un graphe G, la liste des degré L et un sommet v renvoie la liste L' des degrés dans le graphe où v et tous ses voisins ont été supprimés.

Remarque: On ne met pas à jour G !

Question 6.d.
Ecrire en Python l'algorithme glouton GloutonSansConflit(L) qui étant donné un graphe donné sous forme de matrice d'adjacence renvoie un ensemble sans conflit obtenu à l'aide de l'algorithme glouton décrit ci-dessus.

Remarque: Supprimer réellement les sommets d'un graphe est assez technique (il implique renommage des sommets...etc...). Comment faire pour "simuler" une suppression de sommets?

Question 7.a.
Que renvoie votre programme pour un pour un cycle à n sommets, un chemin à n sommets, une clique à n sommets? (Commencez par $n = 4$)
Prouver la cohérence de ces résultats.

Question 7.b.
Evaluer l'algorithme sur la base de données de graphes données sur la page du cours.

Question 7.c.
Quelle est la complexité (théorique) de votre algorithme? Comment évolue votre temps d'éxecution

en fonction du nombre de sommets (avec les graphes évalués dans la fonction 7.b.). Est-ce cohérent?

Pour vérifier que les algorithmes de la question précédente sont exacts, on se propose de coder la fonction suivante qui vérifie la cohérence du résultat:

Question 8.
Ecrire en Python une fonction CheckIndependant(L,X) qui étant donné un graphe G donné sous forme de matrice d'adjacence et ensemble X de sommets vérifie que X est un ensemble indépendant.
(Que renvoie votre programme sur des exemples simples?)

Vérifiez que les ensembles renvoyés par les questions précédentes sont bien des ensembles indépendants.

Question 9.
On peut se demander si l'algorithme glouton (Algorithme 1) renvoie toujours une solution optimale.
a. Proposer un graphe où la solution renvoyée n'est pas optimale.
b. Généraliser ce graphe pour montrer que la solution optimale peut être arbitrairement éloignée de la solution renvoyée par l'algorithme glouton.

4.3 Programmation linéaire

On va utiliser la bilbiothèque PuLP de Python pour résoudre des programmes linéaires. On peut trouver la documentation en ligne de la bibliothèque sur: `https://coin-or.github.io/pulp/`.

Question 10.a.
Écrire une fonction IndependentMax(G) qui étant donné un graphe sous forme de matrice d'adjacence résout le problème de l'indépendant maximum donné à la Question 3. (Que renvoie votre algorithme pour un chemin de taille 5, une cycle de taille 5, une clique de taille 5?)

Question 10.b.
Exécuter l'algorithme sur les graphes fournis. Quel est le temps d'execution pour résoudre les instances? (Si nécessaire rajouter un timeout à 2mn).

Question 10.c
Comparer la qualité des solutions avec l'algorithme glouton. Et les temps d'exécution?

5 Réalisation de toutes les tâches (*)

Le but est maintenant non pas de réaliser le nombre maximum de taches en une étape mais de réaliser toutes les tâches (on veut faire toutes les tâches de maintenance sur le réseau). Autrement dit:

- On désire minimiser le nombre total d'étapes auquel toutes les tâches sont effectuées.
- A chaque étape on réalise un ensemble de tâches sans conflits.

Appelons ce problème Π.

Question 11. (*)
Proposer un algorithme pour résoudre Π en utilisant un algorithme pour calculer un indépendant de taille maximum.

Question 12.(*)
Implémenter cet algorithme COLORATION1(G).

Question 13.(*)
Prouver qu'il existe des cas où cet algorithme n'est pas optimal. Autrement dit, proposer un graphe G où COLORATION1(G) ne renvoie pas une solution optimale.

Question 14.(*)
Proposer un algorithme de programmation linéaire en nombre entier qui étant donné un graphe et un entier k trouve une façon de réaliser toutes les tâches en k étapes si elle existe.

Question 15.(*)
Implémenter cet algorithme dans COLORATION2(G).

Question 16. (***)
Proposer un algorithme de programmation linéaire (avec un nombre exponentiel de contraintes) qui résout le problme Π (sans que k soit donnée en entrée).