Dipayan Chakraborty
Florent Foucaud
Malika More
Adrien Wohrer

Université Clermont Auvergne
Méthodes d'optimisation
BUT Info 2A
2023-2024

TP2 : Algorithmes pour la couverture par sommets

Version avec Solution

Dans ce TP nous allons étudier en pratique le problème de la couverture par sommets de coût minimum pour les graphes non orientés. Ce problème a été présenté en cours magistral. On a un graphe $G = (V, E)$ et une fonction de coût $c : V \to \mathbb{N}$ qui assigne à chaque sommet un coût entier (positif ou nul). Le but est de sélectionner un ensemble S de sommets de G tel que chaque arête a au moins une de ses deux extrémités dans S. Par ailleurs on cherche à minimiser le coût total de la solution, c'est-à-dire minimiser la somme $\sum_{v \in S} c(v)$. Voir la Figure 1 pour un exemple (où les nombres repésentent les coûts des sommets).

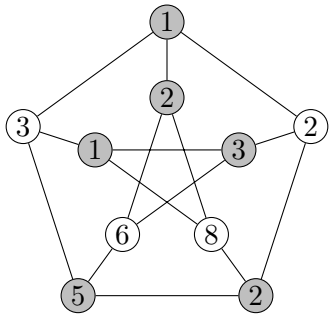

Figure 1 – Les sommets grisés forment une couverture par sommets de coût 14.

On va utiliser Sagemath pour coder et comparer différents algorithmes qui calculent une couverture par sommets (voir en bas du document pour quelques exemples de syntaxe Sagemath pour les PL et les graphes). Pour un graphe G, la fonction de coût des sommets, peut être représentée en Python par un dictionnaire avec comme clés, les sommets et comme valeurs, les coûts.

Pour tester vos algorithmes, vous pouvez créer des graphes. Sagemath dispose de fonctions pour cela. Par exemple, `G=graphs.PetersenGraph()` génère le fameux graphe de Petersen de la Figure 1.
`G=graphs.RandomGNP(20,0.2)` génère un graphe aléatoire à 20 sommets où chaque arête est présente avec probabilité $0.2 = 1/5$. Vous pouvez consulter la bibliothèque de création de graphes de Sagemath `https://doc.sagemath.org/html/en/reference/graphs/sage/graphs/graph_generators.html` pour d'autres exemples.

Vous pouvez générer un dictionnaire de coûts aléatoires `c` dont les clés sont les sommets du graphe et les valeurs sont les coûts. On peut utiliser la fonction `randrange(1,100)` qui renvoie un entier entre 1 et 100 au hasard. Pour pouvoir le réutiliser, vous pourrez par exemple coder une fonction `randomDico(G)` qui renvoie le dictionnaire aléatoire pour le graphe `G`.

Si vous avez une liste de sommets S (votre solution par exemple) d'un graphe G vous pouvez afficher le graphe en colorant les sommets de S d'une couleur spécifique, par exemple en rouge :
`G.plot(vertex_colors={'red':S})`.

Exercice 1 (Algorithme exact par un PLNE)**.**

Coder et tester une fonction `CS_PLNE(G,c)` qui prend en entrée un graphe et son dictionnaire de coûts, et renvoie une solution pour la couverture par sommets (sous forme de liste par exemple).

Cette solution sera calculée en définissant le PLNE du problème, défini de la façon suivante (avec une variable x_v pour chaque sommet v). Le graphe est noté $G = (V, E)$ où V est l'ensemble des sommets et E est l'ensemble des arêtes.

$$
\begin{array}{lllll}
\text{minimiser :} & \sum_{v\in V} x_v c(v) & & & \\
\text{tel que :} & x_u + x_v & \geq & 1 & \forall uv \in E \\
& x_v & \leq & 1 & \forall v \in V \\
& x_v & \geq & 0 & \forall v \in V \\
& x_v & \in & \mathbb{N} & \forall v \in V
\end{array}
$$

Voir en bas du document pour la syntaxe d'un PLNE dans Sagemath.

Notez que cette méthode va calculer la solution optimale. Par contre, cet algorithme prend un peu de temps car résoudre un PLNE n'est pas du tout anodin. Jusqu'à quelle taille de graphes obtient-on la solution en un temps raisonnable (disons, quelques secondes) ? (Essayer avec un nombre de sommets de 50,60,70... etc)

Solution.

```
def randomDico(G):
    c={}
    for v in G.vertices():
        c[v]=randrange(1,100)
    return c

def coutSolution(S,c):
    res = 0
    for x in S:
        res += c[x]
    return res

def CS_PLNE(G,c):
    p = MixedIntegerLinearProgram(maximization=False)
    vars = p.new_variable(integer=True, nonnegative=True)

    for v in G.vertices():
        p.add_constraint(vars[v] <= 1)

    for e in G.edges():
        p.add_constraint(vars[e[0]] + vars[e[1]] >= 1)

    p.set_objective(p.sum(vars[x]*c[x] for x in G.vertices()))

    opt = p.solve()
    R = []
    for x in G.vertices():
        if p.get_values(vars[x])==1:
            R.append(x)
```

```
   return R

G=graphs.RandomGNP(100,.2)#graphe aléatoire à 100 sommets et densité 1/5
c=randomDico(G)
R1=CS_PLNE(G,c)
print("PLNE:",len(R1),coutSolution(R1,c),R1)
```

Sur mon ordinateur personnel, à 90 sommets, on attend environ 1 seconde. A partir de 120 sommets environ, on doit attendre quelques secondes.

Exercice 2 (Algorithme d'approximation par relaxation linéaire)**.**

On va maintenant calculer une autre solution, mais cette fois on va utiliser la méthode de *l'arrondi de la relaxation linéaire* du PLNE vue en cours, qui renvoie une solution au plus 2 fois plus coûteuse que l'optimum. Pour cela on résoud d'abord le PL qui a la même formulation que le PLNE de l'exercice précédent, sauf qu'on ne rajoute pas la contrainte "$x_v \in \mathbb{N}$" (les variables sont non-entières).

Une fois le PL résolu, on va "arrondir" le résultat : pour chaque variable x_v, si $x_v \geq 1/2$ on prend v dans la solution, et si $x_v < 1/2$ on ne prend pas v. Cette méthode renvoie une solution dont le coût est au pire deux fois l'optimum, mais le temps de calcul devrait être bien plus rapide.

Coder et tester une fonction `CS_arrondiPL(G,c)` qui prend un graphe et un dictionnaire de coûts en entrée, et qui renvoie une solution obtenue par cette méthode.

Remarque. Bien souvent, si la solution est grande, cette méthode a tendance à renvoyer tous les sommets. Sur certains graphes (notamment les graphes bipartis) le PL classique renvoie déjà une solution à valeurs entières, et donc optimale. Ces deux cas ne se présentent pas sur l'exemple de la figure suivante.

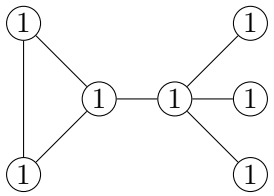

FIGURE 2 – Un graphe intéressant pour tester la relaxation linéaire.

Solution.

```
def CS_relaxPL(G,c):
    p2 = MixedIntegerLinearProgram(maximization=False)
    vars = p2.new_variable(integer=False, nonnegative=True)

    for v in G.vertices():
        p2.add_constraint(vars[v] <= 1)

    for e in G.edges():
        p2.add_constraint(vars[e[0]] + vars[e[1]] >= 1)
```

```
    p2.set_objective(p.sum(vars[x]*c[x] for x in G.vertices()))

    opt = p2.solve()

    R = []
    for x in G.vertices():
        if p2.get_values(vars[x])>=0.5:
            R.append(x)

    return R

R2=CS_relaxPL(G,c)
print("relaxation:",len(R2),coutSolution(R2,c),R2)
```

Exercice 3 (Algorithme heuristique glouton)**.**

On va maintenant utiliser une des heuristiques gloutonnes vues en cours, qui sélectionne à chaque étape le sommet qui couvrira le plus de nouvelles arêtes (en pondérant par son coût) :

- $S = \emptyset$
- Tant que toutes les arêtes ne sont pas couvertes :
  - Calculer le "score" de chaque sommet v : c'est le ratio "nombre d'arêtes non couvertes par S qui seraient couvertes par v" / $c(v)$.
  - Choisir un sommet v qui maximise le score
  - $S = S \cup \{v\}$
- Renvoyer S

Coder et tester cette heuristique dans une fonction `CS_greedy(G,c)` qui renvoie la liste des sommets de la solution obtenue.

Solution.

```
def CS_greedy(G,c):
    solution = []
    aretes_couvertes = []
    while len(aretes_couvertes) < len(G.edges()):
        score_max = -1
        prochain_sommet = None
        for v in G.vertices():
            if not v in solution:
                score = 0
                for e in G.edges_incident([v]):
                    if not e in aretes_couvertes:
                        score += 1
                score = float(score / c[v])
                if score > score_max:
                    score_max = score
                    prochain_sommet = v
        solution.append(prochain_sommet)
        for e in G.edges_incident([prochain_sommet]):
            if not e in aretes_couvertes:
                aretes_couvertes.append(e)
    return solution

R3=CS_greedy(G,c)
print("glouton:",len(R3),coutSolution(R3,c),R3)
```

Exercice 4 (Comparaison des algos)**.**

On peut maintenant comparer la qualité et la performance en temps des différents algorithmes. Pour mesurer le temps de calcul dans une feuille de calcul Sagemath, on peut utiliser :
`timeit.eval("mafonction()",number=1)` (l'argument "number" est le nombre de fois que le même code sera exécuté : si on l'exécute plus qu'une fois il nous affichera le meilleur temps obtenu).

Testez vos différentes fonctions sur une série de graphes de plus en plus grands (par exemple 10,20,50,100,150 sommets...). Pour chaque graphe, lancer chaque fonction implémentée, notez le temps et la qualité de la solution, puis remplissez un tableau qui ressemblera à celui-ci (sur une feuille de papier ou dans un traitement de texte) :

		algo PLNE		algo relax linéaire		algo glouton	
type de graphe	# sommets	coût solution	temps de calcul	coût	temps	coût	temps
aléatoire	50	2.5	60	3.5	30	4.3	20

Solution.

```
for n in [10,20,50,100,200]:
  print("test avec ",n," sommets:")
  G=graphs.RandomGNP(n,.2)
  c=randomDico(G)
  timeit.eval("CS_PLNE(G,c)",number=3)
  R1=CS_PLNE(G,c)
  print("PLNE:",len(R1),coutSolution(R1,c),R1)
  timeit.eval("CS_relaxPL(G,c)",number=3)
  R2=CS_relaxPL(G,c)
  print("relaxation:",len(R2),coutSolution(R2,c),R2)
  timeit.eval("CS_greedy(G,c)",number=3)
  R3=CS_greedy(G,c)
  print("glouton:",len(R3),coutSolution(R3,c),R3)
```

Exemples de syntaxe Sagemath

Sagemath est très pratique pour manipuler des graphes, et s'interface avec des solveurs de PL. Pour le lancer, écrire la commande `sage -n`. Cela lance une interface (basée sur `jupyter notebook`) dans votre navigateur. Vous pouvez ensuite créer une nouvelle feuille de calcul en cliquant sur `Nouveau`→`Sagemath 9.x`. Ensuite on code en Python.

On peut définir et résoudre un PL dans Sagemath de la façon suivante :

```
p = MixedIntegerLinearProgram(maximization=False)            #crée un PL de minimisation
vars = p.new_variable(integer=True, nonnegative=True)        #vars est un dictionnaire qui
                                                              contiendra les variables du PL
                                                              ici ce sont des variables
                                                              entières et non-négatives
p.set_objective( p.sum(3*vars[i] for i in [0,1]) + vars[2] )   #fonction objectif
p.add_constraint( vars[0] + 2*vars[1] >= 4 )                 #contraintes
p.add_constraint( 5*vars[2] - vars[1] >= 8 )
opt = p.solve()                                              #résoudre le PL
print(opt)                                                   #afficher la valeur optimale
print(p.get_values(vars))                                    #afficher la valeur
                                                              des variables
print(p.get_values(vars[x]))                                 #afficher la valeur
                                                              de la variable x
```

Cela correspond au programme linéaire suivant :

$$
\begin{array}{lrcrcrcl}
\text{minimiser} & 3vars[0] & + & 3vars[1] & + & vars[2] & & \\
\text{tel que} & vars[0] & + & 2vars[1] & & & \geq & 4 \\
 & & & -vars[1] & + & 5vars[2] & \geq & 8 \\
 & vars[0] & & & & & \geq & 0 \\
 & & & vars[1] & & & \geq & 0 \\
 & & & & & vars[2] & \geq & 0 \\
 & vars[0] & & & & & \in & \mathbb{N} \\
 & & & vars[1] & & & \in & \mathbb{N} \\
 & & & & & vars[2] & \in & \mathbb{N}
\end{array}
$$

Pour créer et manipuler un graphe non orienté dans Sagemath on peut écrire : [1]

```
G = Graph(3)                    #Crée un graphe à 3 sommets (nommés 0,1,2 par défaut)
G.add_edge(1,2)                 #Ajoute l'arête 1-2
for e in G.edges():             #boucle sur la liste des arcs de G
    print(e[0],e[1])            #affiche l'origine, la destination de l'arête e
for v in G.vertices():          #boucle sur la liste des sommets de G
    print(G.neighbors(v))       #affiche la liste de tous les voisins de v
    print(G.edges_incident([v])) #affiche la liste des arêtes qui touchent v
```

1. Voir `https://doc.sagemath.org/html/en/reference/graphs/index.html` pour plus de fonctions, si nécessaire.

Rappel : structures de données en Python

En Python, une *liste* est un tableau dynamique qu'on peut manipuler de la façon suivante.

```
L = [1,4,"toto"] #création d'une liste avec 3 éléments
L.append(2)      #ajout de 2 en fin de L
L.insert(3,"x")  #insère "x" en 4e position de L
print(len(L))    #affichage de la longueur de L
print(L[1])      #affichage du deuxième élément de L
print(L[-11])    #affichage du dernier élément de L
if 3 in L:       #test si 3 est dans L
for a in L:      #boucle sur les éléments de L
L.remove("y")    #supprime toutes les occurences de "y" dans L
L.pop(2)         #supprime L[2]
L.clear()        #supprime tous les éléments de L
```

Un *dictionnaire* est une structure de données permettant d'associer des clés à des valeurs, on le manipule de la façon suivante.

```
dico = {2 : "toto", 5: "titi", "x": "tata"} #création d'un dictionnaire
dico = {}                                   #création d'un dictionnaire vide
dico[5] = "tutu"                            #mise à jour de l'élément de clé 5
dico[1] = "tete"                            #ajout d'un élément de clé 1
dico.keys()                                 #liste des clés
dico.values()                               #liste des valeurs
dico.pop(2)                                 #suppresion du couple clé/valeur de clé 2
```