CHAPITRE 2 : CLOTURE TRANSITIVE D'UNE RELATION BINAIRE

Michaël PÉRIN – mises à jour Patrick LOISEAU

February 25, 2018

Contents

1 Définition et interprétation

1.0.1 Définition : La cloture transitive de R : AxA est la plus petite relation transitive qui contient la relation R.

La cloture transitive de R est notée Rˆ+ :

1. R+ contient R
2. R+ est transitive
3. toute autre relation qui vérifie 1 et 2 contient plus d'arcs que R+

1.0.2 Construction de la cloture transitive d'une relation R sur AxA

Pour constuire R+, la cloture transitive de R, on crée
un arc a -(R+)-> a' dans R+ pour chaque chemin a -R-> a1 -R-> a2 -R-> ... -R-> a' dans R

1.0.3 Chemin et composition

- La composition R o R = R^2 donne les chemins de longueur 2
- R^1 correspond aux chemins de longueur 1, ie. aux arcs directs, c'est donc R
- R^n donne les arcs correspondants aux chemins de longueur n
- R+ donne les arcs correspondant à tous les chemins de longueur quelconque.
- R+ = R union R^2 union R^3 union ... = Union i>=1 R^i
- Si le domaine A de la relation est fini, de taille n alors R+ = R^n

2 Algorithme de construction de la cloture transitive

La cloture transitive de R est une relation sur AxA notée R+, définie par :

R+ = R : chemins de longueur 1 union (R o R) : chemins de longueur 2 union (R o R o R) : chemins de longueur 3 union ...

= $\mathrm{Union}_{i>0}\ R^{i}$

2.0.1 principe de calcul de la clotûre transitive d'une relation à l'aide de suites récurrentes

On définit

1. r_i = relation formée des chemin de longueur i constitués d'arcs de R = R o R o ... o R (i fois) = R^{i}

   r_i est définie par une suite récurrente

   / $r_1 = R \setminus r_{i+1} = r_i$ o R

2. La clotûre transitive de R, notée R+ est la relation formée des chemins de longueur i>=1

   R+ est définie par une suite récurrente u_i

   où u_i représente $\mathrm{Union}_{1<=l<=i}\ R^{l}$ ie. les chemins de longueur *l* compris entre 1 et i.

   / $u_1 = r_1 \setminus u_{i+1} = u_i$ union r_{i+1}

   le calcul des termes de u_i s'arrête lorsqu'on atteint un rang i tel que $u_{i+1} = u_i$

Avant d'écrire l'algorithme, ouvrons une parenthèse afin d'étudier comment on programme le calcul d'une suite récurrente

3 Calcul des termes d'une suite récurrente

Considérons une suite récurrente u_i définie par :

u_1 = terme1 u_2 = terme2 u_i = f ($u_{i\text{-}1}$, $u_{i\text{-}2}$) où f représente un calcul utilisant $u_{i\text{-}1}$ et $u_{i\text{-}2}$

3.0.1 Exemple : la suite de Fibonacci

est définie par

/ $u_1 = 1$

$$u_2 = 1$$

\ $u_i = u_{i-1} + u_{i-2}$

3.0.2 Les indices des mathématiques correspondent à des cases de tableaux en informatique.

Les variables mathématiques u_1, u_2, u_3, ... correspondent aux cases u[1], u[2], u[3], ...

La fonction suivante calcul le nième terme de la suite u_i :

def nieme(n): u = [1, 1] for i in range(2, n): tmp = u[i-1] + u[i-2] u.append(tmp) return u[n-1]

ou

def $\text{nieme}_{\text{bis}}$(n): u = [4] * n u[4] = 1 u[1] = 1 for i in range(2, n): u[i] = u[i-1] + u[i-2] return u[n-1]

3.0.3 Optimisation

Pour calculer le 1000 ème terme de la suite u_i, la fonction précédente a besoin d'un tableau u[1..1000] et garde en mémoire les 1000 termes de la suite alors que

- on ne s'intéresse qu'au 1000 ième et pas aux termes précédents
- pour calculer u_{1000} on a besion de faire f(u_{999},u_{998}) mais on ne se sert plus des autres termes : ceux avant u_{998}, on pourrait donc les effacer !

Conclusion : on pourrait libérer les cases mémoires du tableau dés qu'on en a plus besion pour les calculs suivants.

---

[1]DEFINITION NOT FOUND.
[2]DEFINITION NOT FOUND.
[3]DEFINITION NOT FOUND.
[4]DEFINITION NOT FOUND.

3.0.4 Exemple :

Au départ on a besoin de u^1 et u^2 pour calculer u^3 puis on a besion de u^2 et u^3 pour calculer u^5 on peut donc libérer u^1 puis on a besion de u^3 et u^5 pour calculer u^6 on peut donc libérer u^2 puis on a besion de u^5 et u^6 pour calculer u^7 on peut donc libérer u^3 etc…

3.0.5 Remarque :

On constate qu'à chaque étape on a en fait besion de trois cases uniquements :

u[i-2] et u[i-1] dont on a besoin pour calculer u[i]

c'est logique puisque la suite définie $u_i = f(\ u_{i-1},\ u_{i-2}\ )$

3.0.6 L'idée :

Pour optimiser le programme on va utiliser un tableau avec seulement trois cases et les réutiliser au fur et à mesure que le calcul avance.

On garde quasiment le même programme qu'avant mais on considère des indices i, i-1, i-2 **modulo 3**.

def $nieme_{bis2}$(n): u = ^{4} * 3 u^4 = 1 u^1 = 1 for i in range(2, n): u[i % 3] = u[(i-1) % 3] + u[(i-2) % 3] return u[(n-1) % 3]

3.0.7 Remarque :

Pour un tableau de taille T, utiliser des indices modulo T permet d'être sûr de ne pas dépasser la taille du tableau puisque *modulo T* lorsqu'on dépasse T-1 on retourne à 0 :

0 -+1-> 1 -+1-> 2 -+1-> …. -+1-> T-1 -+1-> 0 -+1-> etc…

C'est une technique bien connue des informaticiens. Ça évite des bugs de dépassement de tableaux.

---

[5] DEFINITION NOT FOUND.
[6] DEFINITION NOT FOUND.
[7] DEFINITION NOT FOUND.

4 Implantation en C de la construction de la clotûre transitive d'une relation

4.0.1 Au TP3 vous écrirez le programme qui constuit la clotûre transitive d'une relation R

en calculant les termes des suites u_i et r_i définies par

/ $u_1 = r_1$ et / $r_1 = R$ \ $u_{i+1} = u_i$ union r_{i+1} \ $r_{i+1} = r_i$ o R

jusqu'à atteindre un certain rang i tel que $u_{i+1} = u_i$

ie. que les chemins de longueur i+1 n'ajoutent pas d'arcs en plus par rapport aux chemins de longueur <= i

On aura alors :

1. u_i contient R En effet, u_i contient u_{i-1} qui contient u_{i-2} qui contient ... qui contient $u_1 = R$

2. u_i est transitive La preuve s'appuie sur le fait que $u_{i+1} = u_i$ signifie que R^{i+1} est inclus dans R1 union R2 union ... union R^i

/!\ Chaque terme des suite r_i et u_i est une relation sur AxA

5 Application de la cloture transitive : le calcul des distances minimales entre les villes (algorithme de Floyd-Warshall)

On considére l'ensemble V des villes et la relation R qui indique les villes voisines c'est à dire les villes qui sont limitrophes. Mais au lieu de représenter R par un prédicat

R : V x V –> Bool qui se contenterait d'indiquer si deux villes sont limitrophes

on considére que R est une fonction qui donne en plus la distance entre deux villes

R : V x V –> Km

En réalité on représente R par une matrice dont les coefficients sont les distances directes entre deux villes limitrophes (cad. sans passer par une ville intermédaire) .

R possède les propriétés suivantes :

- R[v][v] = 0 km quelle que soit $v \in V$

- R[v1][v2] = R[v2][v1] quelles que soient v1,v2 $\in$ V car si on peut aller de v1 à v2 on peut faire le trajet en sens inverse de v2 à v1 avec le même nombre de kilomètres.

- R[v1][v2] = k km avec k > 0 si v1 et v2 sont limitrophes et distances de k km

- R[v1][v2] = +infini si il est impossible de rejoindre v2 depuis v1 par la terre, par exemple R[Grenoble][Tokyo] = +infini

5.0.1 Le système Mappy ou ViaMichelin

cherche à calculer le chemin le plus court d'une ville V à une autre ville V' en passant par des villes intermédiaires.

Pour cela il considère tous les chemins possibles de V à V' :

- ceux de longueur 1, s'ils existent, ie. si les villes sont limitrophes

- ceux de longueur 2, s'ils existent, ie. passent par 1 ville intermédiaire

- ceux de longueur 3, s'ils existent, ie. passent par 2 villes intermédiaires

- etc

On comprend donc que Mappy ou ViaMichelin ont besoin de constuire la cloture transitive de la relation R !

5.0.2 Il suffit de calculer R+

Pour résoudre la question du plus court chemin il suffit donc de constuire R+. La case *R+[V][V']* donnera la longueur du plus court chemin de V à V'.

On connait l'algorithme de construction de R+. Il ne reste qu'une chose à déterminer :

5.0.3 Avec quel semi-anneau (ring,somme,zero,produit,unit) doit-on travailler ?

L'instruction essentielle du produit de matrice R*S est la suivante

s = somme(s, produit (R[a][b] , S[b][c]))

Dans notre cas S=R et les a,b,c sont des villes v1,v2,v3. On peut donc réecrire cette instrction :

s = somme(s, produit (R[v1][v2] , R[v2][v3])) et chaque R[v1][v2] = distance en km

v1–(d1 km)–>v2 v2–(d2 km)–>v3 \ / ———(d1+d2 km)———

Le produit (R[v1][v2] , R[v2][v3]) doit donner la distance entre v1 et v3 : c'est donc l'addition des distances

Ainsi dans notre problème il faut choisir produit = addition et unit = 0 (l'élément neutre de l'addition)

s = somme(s, R[v1][v2] + R[v2][v3])

L'opérateur «somme» doit choisir entre la solution précédente *s* et la nouvelle solution *R[v1][v2] + R[v2][v3]*. Il faut donc choisir somme = min et zero = +infini (l'élément neutre du min)

5.0.4 Vérifions que (somme=min, zero=+infini, produit=addition, unit=0) respectent les lois d'un semi-anneau

Attention pour avoir le droit de faire des produit de matrice il faut s'assurer que (somme=min, zero=+infini, produit=addition, unit=0) est bien un semi-anneau c'est à dire qu'il faut vérifier :

1. que la produit se distribue sur la somme ie. produit(x, somme(y,z)) == somme(produit(x,y) , produit(x,z))

   ce qui donne pour notre semi-anneau (min,+) : x + min(y,z) ? min(x+y,x+z) : OK c'est vrai

2. que le zero est bien un élément absorbant du produit ie. produit(zero, x) == zero

   ce qui donne pour notre semi-anneau (min,+) : +infini + x ? +infini : OK c'est vrai !

3. que l'opérateur produit est commutatif ie. produit(x,y) == produit(y,x)

   ce qui donne pour notre semi-anneau (min,+) : x+y ? y+x : OK car + est commutatif

4. que l'opérateur produit est associatif ie. produit(x,produit(y,z)) == produit(produit(x,y),z)

   ce qui donne pour notre semi-anneau (min,+) : x+ (y+z) ? (x+y)+z : OK car + est associatif

5. que l'opérateur somme est commutatif ie. somme(x,y) == somme(y,x)

   ce qui donne pour notre semi-anneau (min,+) : min(x,y) ? min(y,x) : OK car min est commutatif

6. que l'opérateur somme est associatif ie. somme(x,somme(y,z)) == somme(somme(x,y),z)

   ce qui donne pour notre semi-anneau (min,+) : min(x,min(y,z)) ? min(min(x,y),z) : OK car min est associatif

Conclusion Une fois ces vérifications faîtes, le problème du plus court chemin entre deux villes consiste simplement à constuire la cloture transitive de R dans le semi-anneau (somme=min, zero=+infini, produit=addition, unit=0).

En 1h30 au TP3 vous serez en mesure de résoudre ce problème.

Ce qui montre qu'il est utile de passer 1h30 en cours pour étudier un problème afin de le résoudre en 1h30 de TP et 100 lignes de Python plutôt que de passer plus jours à programmer pour obtenir 1000 lignes de code qui ne résolvent pas le problème.

Les mathématiques servent à devenir un bon informaticien plutôt qu'un mauvais programmeur.