USTHB
Faculté d'Electronique et Informatique
Département D'Informatique
Master SII, complexité de calcul

Mardi 12 Janvier 2012

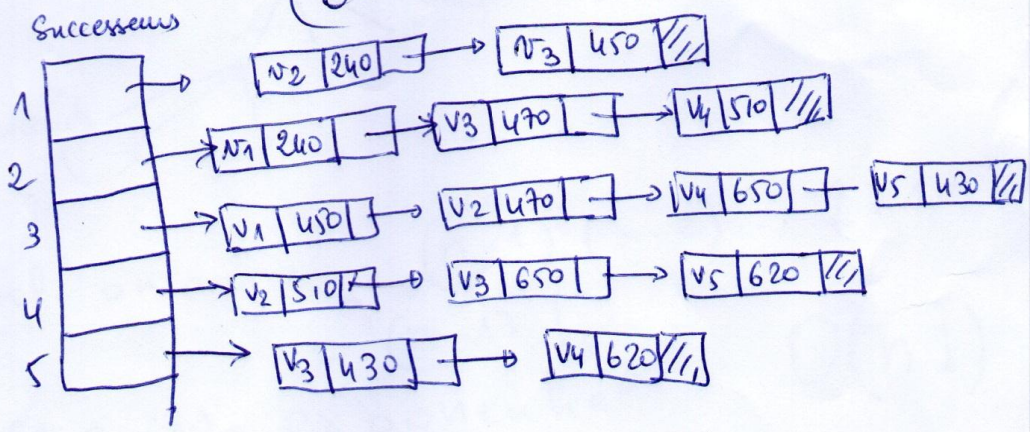
Corrigé de la synthèse

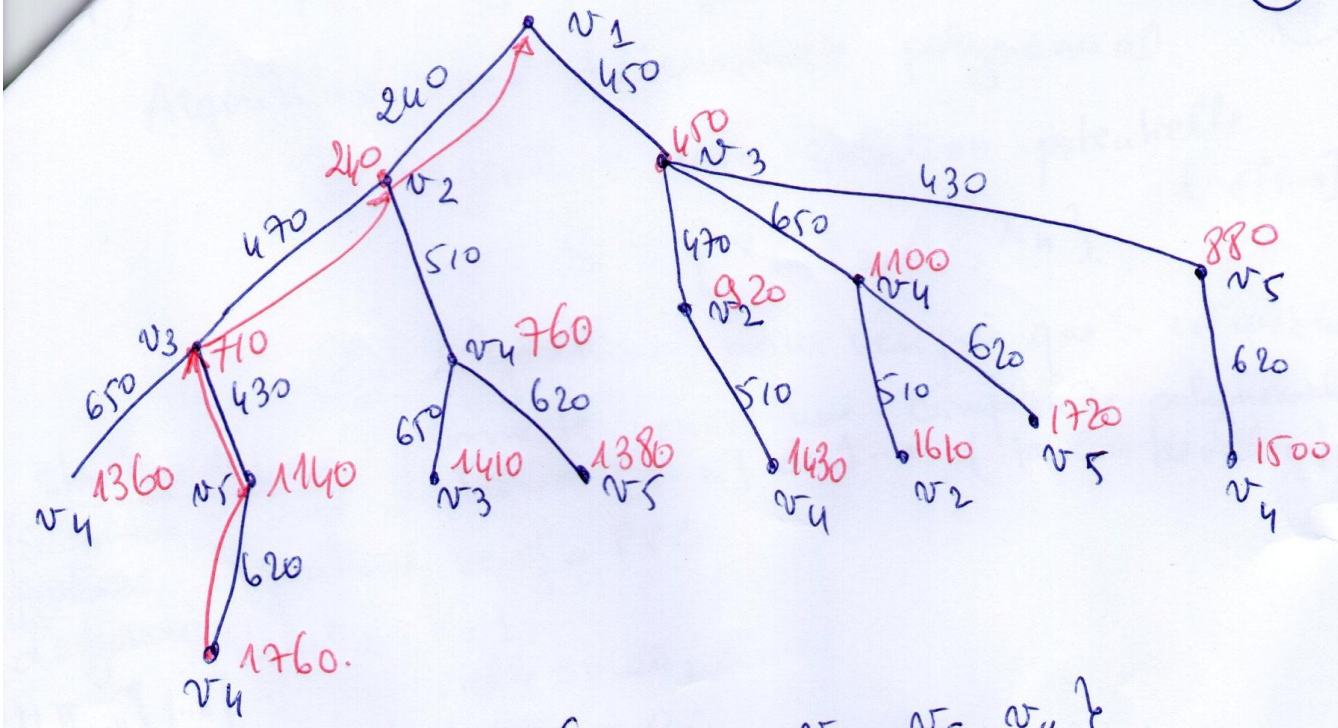
1)

```
Algorithme Voyageur-commerce ;
entrée : /* tableau des successeurs des villes
         successeurs[1..n]
         Ville-départ /* ville de départ
         k
sortie : tournée répondant au problème si elle existe

type   villes : enregistrement V : 1..n ;
                                suiv : ↑villes ;
                fin ;
       élément : enregistrement ville : 1..n ;
                                distance : entier ;
                                VV : ↑villes ;
                                suivant : ↑élément ;
                                précédent : ↑élément ;
                  fin ;
       succ :  enregistrement ville :1..n ;
                                distance : entier ;
                                succsuivant ;
                fin ;
       successeurs : tableau[1..n] de ↑succ ;

var
       x, y, open : ↑élément ;
       s : ↑succ ;
       k, distance : entier ;
       trouve : booléen ;
début
       open = {} ;
       x↑.ville = ville-départ ;
       x↑.distance = 0 ;
```

```
x↑.VV =allouer(villes) ;
x↑.VV↑.suiv = nil ;
x↑.VV↑.V = i ;
x↑.suivant = nil
x↑.précédent = nil ;
insérer(x) ; /* par ordre croisant dans open selon x↑.distance
trouve = faux ;
tant que (open non vide) et (non trouve) faire
début  x = open ; /* le premier élément de la liste open, donc celui avec la
                     distance la plus courte
       open = open↑.suivant ;
       s = successeurs[x↑.ville] ;
       tant que s <> nil faire
       début
            si s n'appartient pas à x.VV alors
            début  distance = x↑.distance + s↑.distance ;
                   si (|x↑.VV| = n-1) et (distance <= k)
                    alors trouve = vrai
                    sinon si (distance <= k) alors
                          début  y = allouer() ;
                                 y↑.ville = s ;
                                 y↑.distance = distance ;
                                 y.VV = rajouter (x↑.VV) ;
                                 y↑.suivant = nil ;
                                 y↑.précédent = x ;
                                 insérer(y) ;
                          fin ;
            fin ;
            s = s↑.succsuivant ;
       fin ;
fin ;
fin .
```

2) (ou matrice d'adjacence)





solution = $\{ v_1, v_2, v_3, v_5, v_4 \}$.

3) Complexité.

pire cas : Graphe complet.

Chaque nœud explore m-1 nœuds si le précédent nœud explore m.

au total on aura : $(n-1)(n-2)\dots 2 \times 1$

$= (n-1)!$

ce qui est exponentiel. $O(n!)$

1). PVC $\in$ NP ? (4)

Algorithme non déterministe polynomial.

① Engendrer une solution potentielle $x_i \in [1..n]$

$S = \{x_1, x_2, \dots, x_n\}$.

② Algorithme pour vérifier que S est une solution correcte avec une complexité polynomiale.

structure de données ← S : tableau [1..n] de 1..n (tableau de villes)

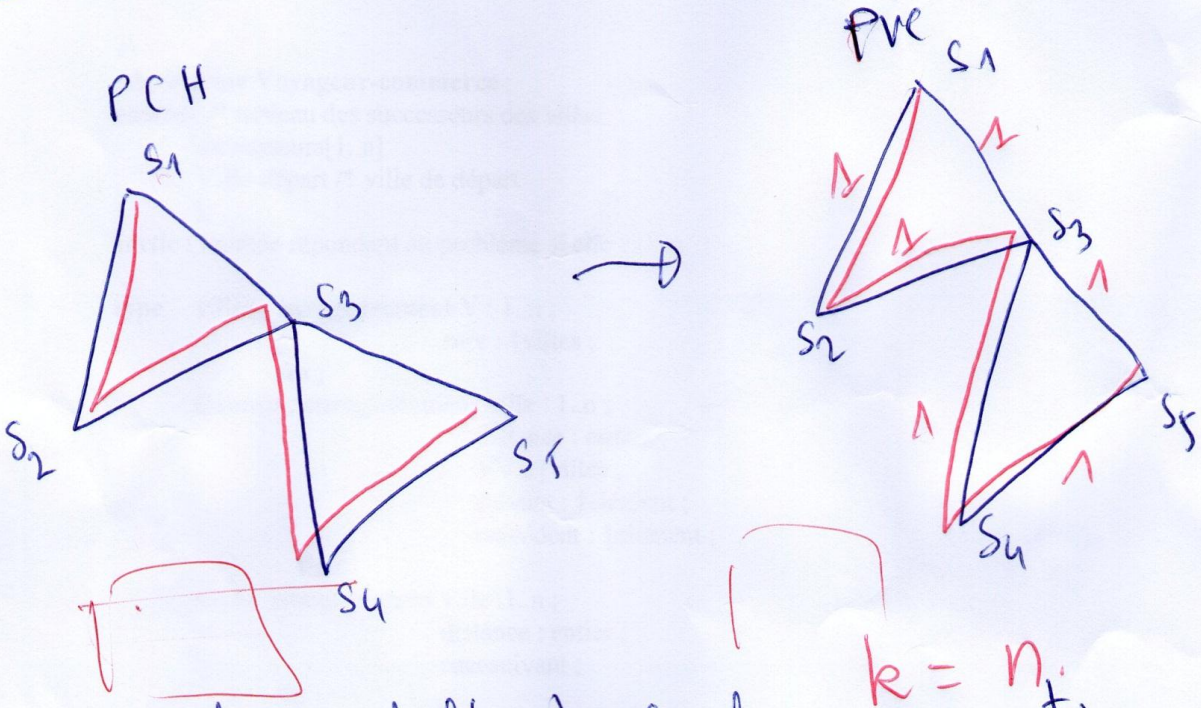
[S = solution
structure d'adjacence
M [1..n] [1..n]]

Algorithme verif-PVC
début
i = 1;
viable = vrai;
distance = 0;
tant que viable et (i ≤ n-1) faire
début
j = i+1;
tant que viable et (j ≤ n) faire
si (S[i] = S[j]) alors viable = faux;
sinon j = j+1;
distance = distance + M[S[i], S[i+1]];
si distance > k alors
viable = faux
sinon i = i+1;
fin
fin

5) PVC est NP-Complet ?

(a) PVC est NP.

(b) Transformation polynomiale entre PCH → PVC.



La construction de l'instance de PVC à partir de celle du CH se fait en $O(n)$. C'est le même ~~tout cas~~ graphe, avec des valeurs égales à 1 associées aux arêtes de plus à chaque solution de PCH, lui correspond une solution de PVC.

~~On associe la valeur 1 aux arêtes~~

La distance parcourue dans tous les cas est inférieure à n.