MASTER 2 - Semestre 1 - 2008

Algorithme du monde réel
Partie 2 en C++

DESENNE Nicolas
DESCOMBE Hervé
SELLIER Xavier

Enseignant : Cyril Gavoille

1 Prototype

Voici l'algo récursif auquel j'ai pensé. La syntaxe n'est pas correcte.
Voilà notre algorithme récursif pour la couverture de sommet :

```
map <key, iterator> graph;

int[] couverture;

int nb_sommet = 0;

function int algo1(var noeud:entiers){
  int couvert = 1;
  Pour chaque fils du sommet  noeud  faire
     couvert *= algo1(nom_du_fils);
  FindePour
  Si couvert == 1 faire
     couvert = 2;
  FindeSi
  Si couvert > 2 faire
     couvert = 1;
     couverture[nb_sommet]=noeud;
     nb_sommet++;
  FindeSi
  Si couvert == 0 faire
     couvert = 1;
     couverture[nb_sommet]=noeud;
     nb_sommet++;
  FindeSi
  Si nb_fils_du_noeud == 0;
    couvert = 0;
  FindeSi
  retourner couvert;
}
```

Exemple de fonctionnement de notre méthode sur notre graphe exemple.
Prenons cet arbre :

0-1-3
|
2-4-7
|
5
|
6

Dans notre variable graphe on aura donc :
[0] -> [1 - 2]
[1] -> [3]
[2] -> [4 - 5]
[3] -> null
[4] -> [7]
[5] -> [6]
[6] -> null
[7] -> null

Déroulement de notre algorithme :

```
algo1(0);
|- algo1(1);
|  |- algo1(3);
|  |  |- retourne couvert = 0;
|  |- couvert = 0;
|  |- couvert = 1;
|  |- couverture[0] = 1:
|  |- nb_sommet = 1;
|  |- retourne couvert = 1;
|- couvert = 1;
|- algo1(2);
|  |- algo(4);
|  |  |- algo(7);
|  |  |  |- retourne couvert = 0;
|  |  |- couvert = 0;
|  |  |- couvert = 1;
|  |  |- couverture[1] = 4;
|  |  |- nb_sommet = 2;
|  |  |- retourne couvert = 1;
|  |- algo(5);
|  |  |- algo(6);
|  |  |  |- retourne couvert = 0;
|  |  |- couvert = 0;
|  |  |- couvert = 1;
|  |  |- couverture[2] = 5;
|  |  |- nb_sommet = 3;
|  |  |- retourne couvert = 1;
|  |- couvert = 1;
|  |- couvert = 2;
|  |- retourne couvert = 2;
|- couvert = 2;
|- couvert = 1;
|- couverture[3] = 0;
|- nb_sommet = 4;
|- retourne couvert = 1;
```

Voici un second algo pour la couverture d'arête cette fois ci :

```
map <key, iterator> graph;

int[] couverture;

int nb_sommet = 0;

function int algo2(var noeud:entiers){
  int couvert = 1;
  Pour chaque fils du sommet  noeud  faire
     couvert *= algo2(nom_du_fils);
  FindePour
  Si couvert == 1 faire
     couvert = 2;
  Sinon
     Si couvert >= 2 faire
       couvert = 1;
       couverture[nb_sommet]=noeud;
       nb_sommet++;
     FindeSi
  FindeSi
  Si couvert == 0 faire
     couvert = 1;
     couverture[nb_sommet]=noeud;
     nb_sommet++;
  FindeSi
  Si nb_fils_du_noeud == 0;
    couvert = 0;
  FindeSi
  retourner couvert;
}
```

P.S : je n'étais pas à poil devant mon ordi lorsque j'ai pensé a cette petite rectification de l'algo (même si l'heure était avancée dans la nuit).

2 Le super algo de fou avec les tableaux

```
int size;

liste feuilles;
liste couverture;
int[] pere;
int[] nbFils;
int racine;

function void algo(){
// on part du principe que la liste ne peut pas être vide.
int i := feuilles.premier();
Si i := racine;
  alors fin algo;

int j := pere[i];
//il n'est pas possible qu'un sommet autre que la racine n'ait pas de
père.
int k := pere[j];
//on retire i de la liste de feuilles.
feuilles.supprimerEnTete();
// i est une feuille, on ajoute son père dans la couverture.
couverture.ajouterEnTete(j);
//On "coupe" l'arrête entre i et son père, j.
pere[j] := 0;
//On "coupe" l'arrête entre j et son père, k.
pere[i] := 0;

Si k != 0
   alors nbFils[k]--;
         Si nbFils[k] = 0
            alors feuilles.ajouterEnTete(k);

algo();

}
```

3 Forêts et couvertures

Soit F une forêt. Soit uv une arête de cette forêt et le degré de v est 1.

Soit U un ensemble de sommets consituant une couverture de F de taille minimale. Si U couvre F, alors U couvre l'arête uv, ce qui implique que u ou bien v appartient à U.

Nous travaillons dans une forêt, une union disjointe d'arbres, ce qui implique que chaque sommet est soit un noeud, soit une feuille. Chaque feuille n'a qu'un seul père, par définition, ce qui signifie qu'un sommet feuille a obligatoirement un degré égal à 1. Considérons maintenant le cas inverse. Si un sommet a un degré égal à 1, alors soit c'est une feuille de l'arbre, soit ça en est la racine. Dans les deux cas, un tel sommet ne peut couvrir que lui-même et son père (ou son unique fils pour le deuxième cas). Le sommet v considéré plus haut, constitue un tel sommet

Supposons que U ne contient pas u. Alors elle contient v, et l'arête uv est couverte. Toutefois, le sommet u possède probablement d'autres arêtes incidentes. Si F n'est pas limitée à l'arête uv, puisque v est de degré 1, alors u possède d'autres arêtes incidentes (du point de vue d'un arbre, u possède d'autres fils et/ou a un père). Imaginons que u possède un autre fils feuille autre que v, que l'on appelle w. Alors pour que l'arête uw soit couverte, il faut que soit u, soit w appartienne à U. Hors, si u n'appartient pas déjà U, alors il faut ajouter un sommet (soit u, soit w) à la couverture, ce qui augmente sa taille.

Supposons maintenant que l'arête uv n'est plus couverte par v, mais par u. Appelons U' cette couverture. Nous n'avons pas modifé la taille de la couverture, donc U' est bien de taille minimale. Reconsidérons maintenant cette arête uw. Dans le cas présent, u couvre désormais cette arête. Il existe donc bien une couverture de taille minimale qui contient u mais pas v.

4 Algorithme linéaire

C'est l'algorithme auquel j'avais pensé (Hervé). Il parcourt les sommets en profondeur une seule fois. Le seul bémol est qu'il faut au préalable avoir parsé le gengraph. Si ècca fait partie de l'algo, ben c'est un peu mort (à moins de l'adapter).

```
couvArbre (r : entier, liste_couv : liste_d_entiers) : entier {

  booleen couvrant := faux ;

  pour chaque fils de r {
    entier s := fils(r);
    si nombre_de_fils (s) = 0  // si s est une feuille
      couvrant := vrai;        // r doit faire partie de la couverture
    sinon
      si (non couvArbre(s), liste_couv)    //Si s n'est pas une feuille, on regarde s'il fa
        couvrant := vrai;                  //Si ce n'est pas le cas, alors r doit couvrir l
  }

  si (couvrant := vrai)
```

```
    ajouter(list_couv, r);
  retourner couvrant;
}
```

5 Algorithme approché sur des graphes

Soit G un graphe connexe non orienté. Soit T l'arbre orienté obtenu par une recherche en profondeur sur G. Si G est connexe, alors $\forall u, v \in V(G), \exists chemindeuav$. Cela implique que tous les sommets feront partie de l'arbre T.

Soit T l'arbre orienté correspondant à une arborescence possible de G. Tout sommet est soit une feuille, soit un sommet. Or, dans un arbre, les relations de père à fils (c'est-à-dire les arêtes entre deux sommets) sont soient des relations de type noeud/noeud, ou bien noeud/feuille (reprécisons que la notion de noeud équivaut à un sommet non feuille). Donc, si l'on retourne l'ensemble des noeuds de l'arbre, comme couverture de cet arbre, alors toutes les relations dans l'arbre (soient toutes les arêtes) sont couvertes. Cet algorithme retourne bien une couverture du graphe G.

5.1 Algorithme basé sur le couplage maximum

5.2 Algorithme basé sur la recherche en profondeur

(Voici l'algorithme auquel j'ai pensé. Je ne l'ai pas implémenté. Je laisse ça à Nico ;-) (Attention, séquence nostalgie, j'ai fait un peut d'exalgo, mais c'est tellement casse-couille que c'est mélangé à du C/C++)

```
parcoursProfondeur(int r, vector<list<int>> & list_succ, vector<bool> & visited):vide
  visited[r] := vrai;
  pour toute arete incidente a r faire
    s := voisin[r];                 //Voir comment on va faire pour connaitre les voisins d
                                    //Peut-etre parser directement ici.
    si (visited[s] = faux) alors
      ajouter(list_succ[r], s);
      parcoursProfondeur(s, list_succ, visited);
    finsi
  finpour
fin


couvProfondeur(vector<list<int>> & list_succ):liste_de sommets(reference)
  liste_couv : liste_de_sommets
  pour chaque sommet r de list_succ
    si (list_succ[r] != vide)
      ajouter(liste_couv, r);
    finsi
  finpour
  retourner liste_couv;
fin
```

(La liste de sommets sera bien sûr une liste d'entiers.)

Pour ce qui est de la complexité, la procédure de parcours en profondeur va être en $O(2 * E(G))$ (*E(G)* étant le nombre d'arêtes du graphe G). En effet, pour un sommet s, on regarde chaque voisin. Si ce voisin (disons v) n'a pas été visité, on explore celui-ci avant d'en finir avec s. Comme on examine chaque voisin de chaque sommet, on va explorer les voisins de v, et on va donc retomber sur s. Comme s a été marqué comme visité, on ne va pas le réexplorer, mais cette suite d'opérations équivaut à visiter deux fois l'arête sv. En appliquant ce raisonnement à toutes les arêtes, on obtient donc une complexité d'ordre $O(2*E(G))$. Hors, dans le cas le plus coûteux où G est un graphe parfait, il va contenir $n-1$ arêtes à visiter pour le premier sommet, puis $n-2$ pour le deuxième, et ainsi de suite jusqu'au dernier sommet. Le nombre d'arêtes correspond alors au calcul de la suite des $n-1$ premiers entiers. On peut alors écrire : $E(G) = (n*(n-1))/2$. On obtient donc une complexité de $O(n*(n-1))$ (puisqu'on parcourt l'équivalent du double du nombre d'arêtes).

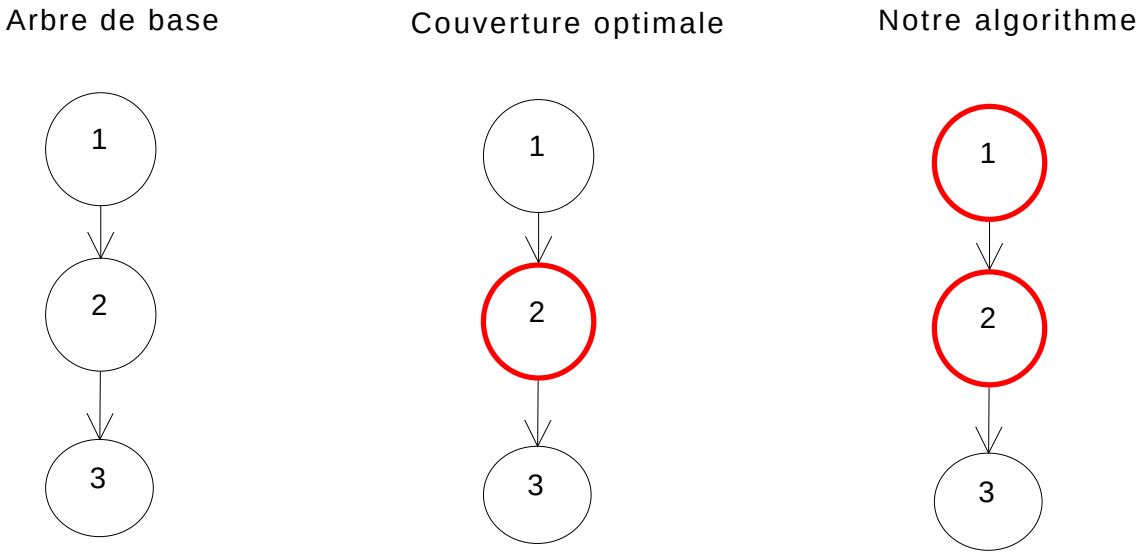
La fonction *couvProfondeur* parcourt le vecteur de sommets (de taille le nombre de sommets, soit$n$), et se contente de mettre dans une liste les sommets non-feuille, qui feront partie de la couverture. Cette fonction a donc une complexité de $O(n)$.

La complexité de l'algorithme est donc de l'ordre de $O(n*(n-1)+n)$, soit $O(n^2)$. (Attention hypothèse dans laquelle on ne compte pas le temps de calcul des voisins de chaque sommet, à moins qu'il soit fait directement dans la procédure de parcours en profondeur.)
(J'avouerai que j'ai un doute du fait d'obtenir un algo en temps polynomial, mais c'est peut-etre ça après tout, puisqu'on est sur des graphes quelconques.)

6 Algorithme approché sur des graphes

Soit G un graphe tel qu'il n'existe qu'une seule feuille. Par conséquent chaque noeud ne possède qu'un seul et unique fils. Nous prendrons un arbre de ce type mais de taille 3.

Si nous appliquons l'algorithme vu en cours nous obtiendrons une couverture optimale, et si nous appliquons notre algorithme nous obtiendrons une couverture deux fois plus grande que la couverture optimale.



On constate que nous prenons les noeuds 1 et 2, alors que l'algorithme de couverture optimale ne prend que le noeud 2. Nous avons donc 2 fois plus de noeuds dans notre couverture que l'algorithme optimale, c'est donc une couverture 2 approché.