MASTER 2 - Semestre 1 - 2008

Algorithme du monde réel
Partie 2 en C++

DESENNE Nicolas
DESCOMBE HervÃ©
SELLIER Xavier

Enseignant : Cyril Gavoille

1 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.

2 Algorithme récursif couverture taille minimale

Le principe de cet algorithme est assez simple. Nous avons une liste de feuilles, une liste contenant les sommets appartenant à la couverture, un tableau contenant les pères des sommets, un tableau contenant le nombre de fils des sommets, et enfin la racine de l'arbre.

Cet algorithme est de complexité linéaire. Cet algorithme n'a pas besoin de savoir quel sommet est voisin d'un autre, il faut juste savoir qui est père de qui. Cet algorithme se finit quand la seule feuille restante est la racine, cet algo va détruire l'arbre au fur et à mesure de son déroulement.

On commence par prendre une feuille, on cherche son grand père. Un sommet a toujours un père, sauf la racine, donc on se passe de la vérification du père. On coupe l'arête entre i et son père, et entre le père et le grand père et on ajoute le père à la couverture. Enfin si k est devenu une feuille on le rajoute á la lise de feuilles.

```
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 Algorithme 2-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.

3.1 Algorithme basé sur le couplage maximum

Le but de cet algorithme est de calculer un ensemble d'arêtes (que l'on appelle M tel qu'aucune d'entre elles n'ait une extrémité en commun. L'autre particularité est qu'on ne peut pas ajouter d'arêtes à cet ensemble sans que la première propriété énoncée juste avant ne devienne fausse. Ainsi, si on ne peut pas rajouter d'arêtes à cet ensemble, cela siginifie que pour chaque arête n'appartenant pas à M, l'une des deux extrémités est incluse dans M. Soit U les extrémités des arêtes appartenant à M. Si une arête appartient à M, alors elle est couverte par ses deux extrémités. Si elle n'y appartient pas, alors elle est couverte par l'une de ses deux extrémités. Donc, U est bien équivalent à une couverture de sommets.

Notre algorithme de couplage, consultable dans le fichier source, se contente de parcourir la liste des arêtes du graphe. Pour une arête donnée, si aucune de ses extrémités n'appartient à la couverture actuelle, alors on les insère dedans. Si l'une des deux extrémités y appartient déjà, alors on n'insère rien du tout. La complexité en temps de calcul est tributaire du nombre total d'arêtes.

3.2 Algorithme basé sur la recherche en profondeur

Nous avons trouvé deux algorithmes qui donnent le même résultat mais de différentes manières. Nous présentons l'algorithme qui suit puisque l'autre est consultable depuis le fichier source.

```
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];
    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
```

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 $O(n^2)$.

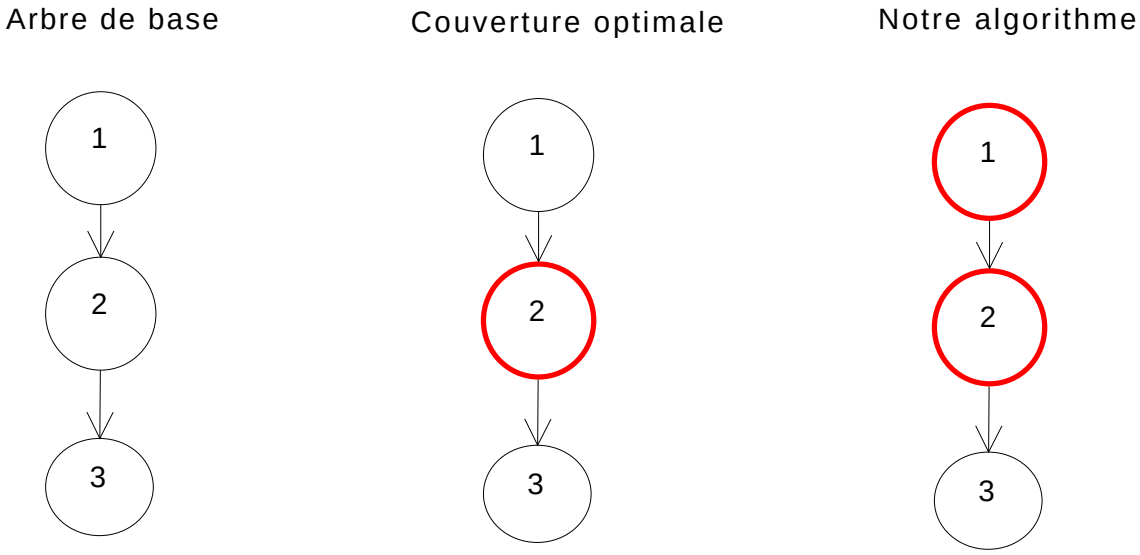
Pour l'algorithme que nous avons implémenté, la complexité est également en O(n^2).

4 Preuve poru la 2-Approximation

4.1 Exemple pour 2-approché

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.



Nous avons orienté le graphe pour donner un sens à notre arbre, mais notre arbre n'est pas orienté.
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.

4.2 Peuve pour 2-approché

4.2.1 Preuve par récursivité pour 2-Approché dans un arbre

Soit G un arbre de taille n. Soit k le nombre de feuille de notre arbre G.

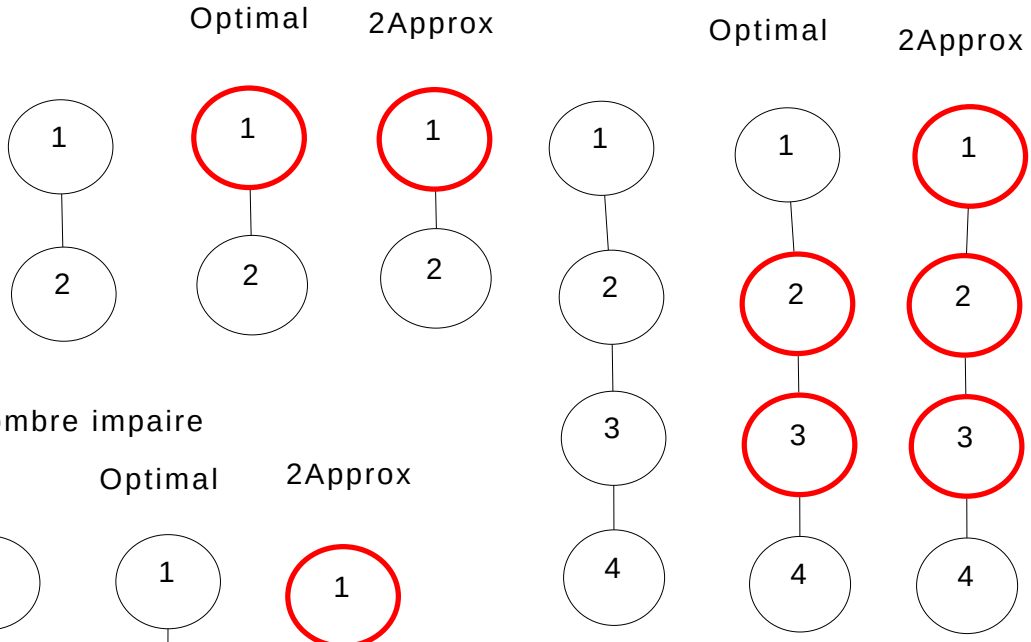
On note O, la taille de la couverture optimale et A la taille de la couverture retourné par notre algorithme.

1. Pour k = 1,

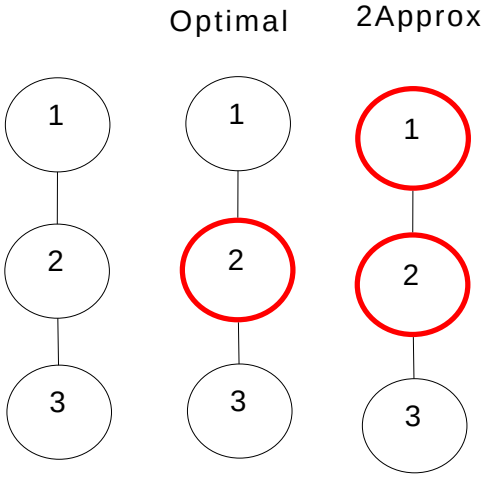
   Si notre arbre n'a qu'une seule feuille, alors chaque noeud n'a qu'un seul fils. Avec notre algorithme nous prendrons n-1 sommets (car k = 1).
   Avec l'algo optimal nous avons deux cas possibles :
   – $\frac{(n-1)}{2}$ si n $\equiv 1(2)$,
   car tous les noeuds de la couverture couvrent chacun deux arêtes.

– $\frac{(n)}{2}$ si n $\equiv 0(2)$,
car tous les noeuds de la couverture couvrent chacun deux arêtes mais une arête est couvert deux fois.

Cas nombre paire



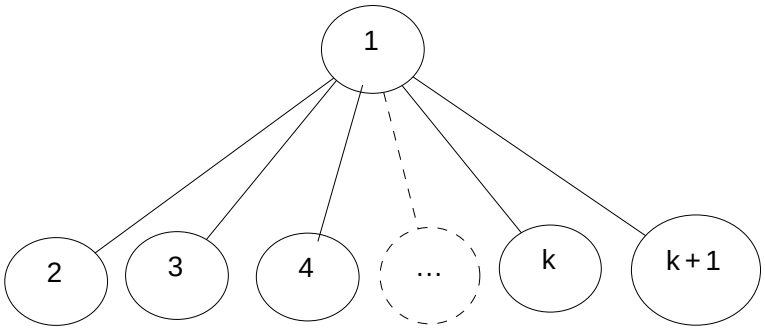
Cas nombre impaire



Donc O * 2 $\geq$ A.

2. On propose d'admettre que ce cas est vrai, autrement dit que O * 2 $\geq$ A pour k fils, pour le même nombre de noeuds (rang n).
3. Montrons que cela est vrai pour k+1 fils, pour le même nombre de noeuds.

   On obtient le graphe suivant :



Nous avons un noeud père et tous les autres sont des racines. Notre couverture optimale est de la même taille que la couverture de notre algo. Donc O * 2 $\geq$ A vrai pour tous les arbres.

4.2.2 Preuve par récursivité pour 2-Approché dans un graphe

Soit G un graphe de taille n possédant $\frac{(n(n-1))}{2}$ arêtes. Le graphe G est donc fortement connexe. On note k, le nombre d'arête que l'on enlève a notre graphe qui restera connexe (mais pas fortement connexe).

1. Pour k = 0,
   Alors nous avons une clique. Notre parcours en profondeur nous retournera un arbre avec une seule feuille.
   D'après la preuve précédente A = n-1 et O = n-1 également. Donc O * 2 $\geq$ A est vrai
2. On propose d'admettre que cela est vrai jusqu'au rang k = $\frac{(n-1)}{2}$. Il est important de noter que notre graphe doit rester connexe, peu importe sa forme.
3. Montrons que cela est vrai pour k = $\frac{n}{2}$.
   Nous obtenons donc un graphe de taille n et possédant n-1 arêtes. Notre graphe est connexe par conséquent, cela est un arbre de taille n. Nous avons déjà fait la preuve que O * 2 $\geq$ A pour un arbre de taille n dans la preuve précédente. Donc cela veut dire que notre preuve est terminée et que O * 2 $\geq$ A.