

Algorithmique Avancée - TD 2 : Corrigé - 13 octobre 2011

Université Paris 7 — BioInformatique — M1

1 Graphes bipartis

Exercice 1 Si un graphe a un cycle de k sommets c_1, \dots, c_k , alors c_i est rouge implique que c_{i+1} est vert, et inversement. Si k est impair c_1 et c_k ont donc la même couleur, et l'arête qui les relie est unicolore. Un biparti ne contient donc pas de cycle impair.

La réciproque est vraie. Montrons par récurrence que tout graphe connexe sans cycle impair est biparti. C'est vrai pour le graphe à 1 sommet.

Supposons-le vrai pour tout graphe à n sommets et montrons-le pour un graphe G à $n + 1$ sommets. Soit x un sommet de G . $G - x$ (graphe obtenu en effaçant x) est sans cycle impair (on ne crée pas de nouveaux cycles en effaçant un sommet) et connexe (dans tout graphe connexe on peut trouver un x tel que $G - x$ est connexe). Par hypothèse de récurrence il est biparti. On colorie donc ses sommets en vert et rouge.

Soit c_1, \dots, c_k un chemin. Si k est impair alors c_1 et c_k ont même couleur, et si k est pair alors c_1 et c_k ont des couleurs différentes. En effet chaque arête étant bicolore on change de couleur à chaque pas. Si x a deux voisins de couleurs différentes, alors x plus le chemin reliant ces voisins (qui existe, puisque le graphe est connexe) forme un cycle de $k + 2$ sommets, de longueur impaire, impossible. Tous les voisins de x ont donc même couleur et on peut colorier x dans l'autre couleur : G est biparti.

Exercice 2

```
Donnée :  $G$  graphe à  $n$  sommets;  
 $C[1..n]$  tableau initialisé à 0;  
 $A$  : file vide;  
début  
  pour  $x$  de 1 à  $n$  faire  
    si  $C[x]=0$  alors  
       $C[x]=1$   
      mettre  $x$  dans  $A$   
      tant que  $A$  non vide faire  
        Extraire  $y$  de  $A$   
        pour tout voisin  $z$  de  $y$  faire  
          si  $C[z]=1$  alors  
             $C[z]=3-C[y]$   
          sinon  
            si  $C[z] \neq C[y]$  alors  
              afficher "pas biparti" ; fin  
        retourner  $C$   
fin
```

Exercice 3 On ne sait rien dire de plus sur le graphe (pas de jolie propriété). De plus le problème est dans la classe des problèmes NP-complets, ce qui veut dire qu'il y a peu d'espoir qu'un algorithme de parcours le résolve (un étudiant qui en écrit un gagne un million de dollars, et ce n'est pas une plaisanterie).

2 Parcours en largeur

Exercice 4 Il suffit de remarquer que chaque arête uv compte deux fois dans la somme : une fois dans $deg(u)$ et une fois dans $deg(v)$. Un arc uv lui ne compte qu'une fois, dans $deg(u)$. Cette preuve devient une récurrence : on ajoute les arêtes (resp. arcs) une à une, à chaque fois m augmente de 1 et la somme de 2 (resp. 1).

Exercice 5 Les initialisations des tableaux sont en $O(n)$. Toutes les autres instructions sont faites en temps constant $O(1)$. Ne reste qu'à compter le nombre de tours de boucle.

On rentre dans le **tant que** n fois au plus, car à chaque fois on marque un sommet et ce marquage est irréversible.

Plutôt que de compter le nombre maximal de fois où l'on rentre dans le **pour** à chaque tour (ce maximum est $n - 1$, degré maximum d'un sommet, on serait donc en $O(n \times (n - 1)) = O(n^2)$) on va compter le nombre **total** de fois : pour chaque sommet u on rentre $deg(u)$ fois exactement, au plus une fois. Au total on fait donc au plus

$$\sum_u deg(u)$$

tours, donc $O(m)$ tours d'après l'exercice précédent.

Donc les lignes $y = \dots$ et $Etat[y] = \dots$ sont faites n fois chacune et le **si** m fois au pire. En additionnant tout on est en $O(n) + O(n) + O(m) = O(n + m)$.

(tournez la page!)

Exercice 6 Il suffit de lancer une routine d’affichage quand on atteint la cible. Elle va de père en père. Attention, cela affiche le chemin à l’envert ! On utilise donc une fonction récursive initiale : la pile inversera cela.

```

AfficheRec(a : sommet cible, b : sommet courant, Pere : tableau)
si a ≠ b alors
  | AfficheRec(a, Pere[b], Pere)
Afficher(b)

BFS(G : graphe, a : sommet départ, b : sommet arrivée)
d[ ] : tableau initialisé à  $-\infty$  ; d[a] = 0
Pere[ ] : tableau initialisé à null
Etat[ ] : tableau initialisé à Non-Atteint ; Etat[a] = Atteint
Atteints : file vide ; MettreEnQueue(Atteints, a)
tant que Atteints non vide faire
  | y = ExtraireTete(Atteints)
  | Etat[y] = Marqué
  | pour tout voisin u de y faire
    | si u = y alors
      | AfficheRec(a, b, Pere)
      | Fin du programme
    | si Etat[u] = Non-Atteint alors
      | d[u] = d[y] + 1
      | Pere[u] = y
      | Etat[u] = Atteint
      | MettreEnQueue(Atteints, u)

```

Exercice 7 Rien de ce qui a été dit ni pour la preuve de correction ni pour l’analyse de complexité ne concerne l’orientation : on a toujours considéré les arêtes uv comme permettant d’aller de l’origine vers la destination, jamais dans l’autre sens. On peut donc les remplacer par des arcs uv et tout reste vrai.

Exercice 8 Là cela ne marche plus. En effet BFS trouve des plus courts chemins en nombre d’arêtes, c’est dans sa preuve de correction. Donc pour un graphe avec deux chemins de a en b , l’un faisant trois arêtes de longueur totale 3 (longueur 1 par arête) et l’autre avec deux arêtes, de longueur totale 4 (2 par arête) il trouvera comme “plus court” le chemin de longueur 4, pas 3. Il nous faut passer à d’autres algorithmes : Bellman-Fort et Dijkstra.

3 Graphes Euleriens

Exercice 9 Un cycle “rentre” autant de fois dans un sommet qu’il en “sort”. Si le cycle est simple (emprunte chaque arête au plus une fois) et passe k fois par un sommet alors il touche $2k$ arêtes. Si il passe par toutes les arêtes du sommet (cas d’un cycle eulerien) alors son degré est $2k$, pair.

Exercice 10 Si eulerien, alors le long du cycle on peut atteindre tout sommet depuis tout autre, le graphe est donc connexe.

Exercice 11 Si la marche stoppe sur un sommet *qui n’est pas le sommet de départ* alors, si on est rentré k fois dans le sommet, on en est sorti $k - 1$ fois (puisque là on ne peut plus en sortir). Comme on a vu toutes les arêtes son degré est $2k - 1$, impair. Ce qui n’est pas possible. Donc on stoppe sur le sommet de départ ($k - 1$ entrées pour $k - 1$ sorties, c’est pair).

Exercice 12 Il suffit de dessiner un graphe à 5 sommets : deux triangles ayant un côté en commun, et de faire une marche qui bloque dans le triangle de gauche.

Exercice 13

```

Eulerien( $G$  : graphe à  $n$  sommets,  $r$  : sommet de départ);
début
   $C_1 = \text{marche}(G,r)$ 
  si  $C_1$  n'est pas un cycle (sommet fin  $\neq r$ ) alors
    STOP : le graphe n'est pas eulerien (sommet fin a degré impair)
  si il reste des arêtes hors de  $C_1$  alors
    Soit  $G' = G - C_1$  (on enlève aussi les sommets déconnectés, de degré 0)
    si il n'existe aucun sommet de  $G'$  apparaissant dans  $C_1$  alors
      STOP : le graphe n'est pas eulerien (pas connexe)
    Soit  $s$  un sommet de  $G'$  apparaissant dans  $C_1$ 
     $C_2 = \text{Eulerien}(G', s)$ 
     $C = \text{Fusionner}(C_1, C_2, s)$ 
  sinon
     $C = C_1$ 
  retourner ( $C$ )
fin
Marche( $G$  : graphe à  $n$  sommets,  $r$  : sommet de départ)
début
   $C = \text{liste vide}$ 
  tant que  $r$  a un voisin  $s$  tel que  $rs$  non marquée faire
     $C = C.\{rs\}$ 
    marquer  $rs$ 
     $r = s$ 
  retourner  $C$ 
fin

```

Exercice 14 Théorème d'Euler : un graphe est eulerien **si et seulement si** il est connexe et sans cycle.

Notre algorithme en est une preuve que si G est connexe sans cycle alors G est eulerien. En effet

1. L'algorithme **termine** car G' est strictement plus petit que G à chaque appel récursif (r a degré non nul donc la marche depuis r enlève au moins une arête)
2. **il retourne un cycle.** Si C_1 n'est pas un cycle il stoppe, mais dans ce cas par l'exercice 6 le sommet d'arrêt a degré impair. Donc C_1 est un cycle ; et la fusion de deux cycles (C_1 et C_2) ayant un sommet en commun (y) est un cycle.
3. **Ce cycle est simple** car une marche passe au plus une fois par arête et ensuite les arêtes sont enlevées en cas d'appel récursif
4. **il contient toute les arêtes.** En effet si une arête est "oubliée" à la fin (pas dans C , donc ni dans C_1 ni dans C_2), c'est qu'elle est dans G' lors de tous les appels. G' n'est donc jamais vide. Or l'algorithme a terminé (et renvoyé C) sans erreur, donc G' est vide, contradiction.
5. On a donc un cycle eulerien

Cette preuve est constructive.