Algorithmique et Programmation **4 Novembre 2009**

TD 5: Graphes

Chargé de TD: Pierre-Alain Fouque *Scribe: Alexandre Thevenet*

5.1 Exercice 1 : Graphe Scorpion

Pour démontrer qu'un graphe est scorpion, le plus simple à faire est d'exhiber les nœud *sting*, *tail* et *body*. Si on ne parvient pas à les trouver, c'est que le graphe n'est pas de type scorpion.

Commençons par lire une colonne de la matrice d'adjacence, et insérons respectivement dans les listes B et S les nœuds qui sont connectés à ce nœud. (n-1 lecture dans la liste d'adjacence)

* Si l'on dénombre 1 seul nœud connecté à celui la, c'est soit qu'il n'est connecté qu'à *body*, soit qu'à *tail* (en supposant le graphe de type scorpion). En lisant la colonne du nœud voisin on trouve donc soit 2 arrêtes (auquel cas on a trouvé *tail*, et donc le nœud en question était *sting* et l'autre devrait être *body*), soit n–2 et alors la seul arrête non reliés à ce nœud (autre que lui même) est *sting*, qui est uniquement reliée à *tail*). (soit $3 * (n - 1)$ lectures de la liste d'adjacence pour vérifier que ces nœuds possèdent bien la propriété voulue)

* De même, si l'on trouve 2 où n–2 arrêtes, on peux trouver *sting*, *body* et *tail* en $3(n - 1)$ lectures de la liste d'adjacence.

* Enfin, dans le cas où on a entre 3 et n–3 arrête, on sait que le nœud *body* est nécessairement dans B, et que *sting* est dans S. De plus, la colonne lue n'étant alors ni *body* ni *sting*, on sais que *tail* n'est pas dans B non plus. On regarde alors tour à tour si le premier élément x de B est relié au premier élément y de S.

- Si oui, alors y n'est pas *sting* (car relié à un nœud qui n'est pas *tail*), et on peux l'enlever de S en étant sur que *sting* soit toujours dans S.

- Si non, alors on enlève x de la liste, puisque soit il n'est pas body, soit il l'est et dans ce cas y est *sting* et "videra" B sans jamais être enlevé.

Ainsi, l'ensemble $\{B + S\}$ ayant n-1 éléments, et perdant un élément à chaque tour, on sais qu'en n-2 tests sur la matrice d'adjacence on aura vidé une des listes B ou S.

- Si S est vide, puisque *sting* est nécessairement dedans, c'est qu'il n'existe pas, et donc que le graphe n'est pas de type scorpion.

- Si B est vide, on sais que, si le graphe est scorpion, alors *sting* est le premier élément de S. On vérifie alors en $3 * (n - 1)$ tests sur la matrice d'adjacence que le graphe est bien scorpion, et on obtient la réponse désirée.

Ainsi, on est donc capable de déterminer si un graphe est scorpion en 5*(n-1) tests sur la matrice d'adjacence (voir un peu moins en gardant certains résultats intermédiaires...), soit en O(n) tests.

5.2 Exercice 2

5.2.1 Question 1

Un graphe acyclique ne contenant pas de boucle, chaque point du graphe peux être vu comme le sommet d'un arbre de nœuds. Ainsi l'algorithme :

ColoriageAcyclique($Parent, noeud, Couleur$) // $Couleur \in \{0, 1\}$
  $noeud.Couleur \leftarrow Couleur$
  for all $V \in noeud.Voisin$ **tels que** $V \neq Parent$ **do**
    ColoriageAcyclique($noeud$, V, $1 - Couleur$)
  end for

Parcourt "l'arbre" des nœuds du graphe en profondeur, et colorie l'intégralité du graphe puisqu'il est connexe.

5.2.2 Question 2

Si un graphe admet un cycle de longueur 3, alors nécessairement les couleurs de chacun des nœuds de ce cycle doivent être différentes de celles des deux autres nœuds, et le graphe n'est donc pas 2-coloriable.

5.2.3 Question 3

Pour voir si un graphe est 2-coloriable, il suffit à priori d'essayer de le 2-colorier (en partant par exemple d'un graphe dont tous les nœuds sont initialisé à la couleur -1). Si on tombe sur un nœud déjà colorié, soit la couleur coïncide, et on continue le coloriage, soit elle diffère et alors le graphe n'est pas 2 coloriable :

Colorier($noeud, Couleur$)
  if nœud.Couleur = -1 **then**
    $noeud.Couleur \leftarrow Couleur$
    for all $V \in noeud.Voisins$ **do**
      if Colorier(V, $1 - Couleur$) = **false then**
        return false// Si un des voisins n'as pas pu être colorié, on arrête tout
      end if
    end for
    return true// on a bien pu colorier ce nœud et ses voisins
  else if nœud.Couleur = Couleur **then**
    return true// le nœud a déjà été bien colorié
  else // Si on a déjà colorié ce nœud de l'autre couleur, le graphe est non-coloriable
    return false
  end if

TestColoriage(*graphe*)
  for all $N \in$ *graphe* **do**
    $N.Couleur \leftarrow -1$
  end for
  for all $N \in$ *graphe* **tels que** $N.Couleur = -1$ **do**
    if (**Colorier**$(N, 0) =$ **false**) **then**
      print `"le graphe n'est pas 2-coloriable"`
      return
    end if
  end for

Cet algorithme parcours donc une fois l'ensemble des arrêtes du graphe, et travaille donc en O(nbr arrête)

5.2.4 Question 4

Un tel graphe est nécessairement $(\Delta + 1)$-coloriable puisqu'il y a toujours assez de couleurs pour colorier un nœud et tous ces voisins avec des couleurs 2 à 2 différentes. En effet soit l'algorithme suivant :

ColorierNœud(*noeud*,Δ)
  for $i = 0$ **to** Δ **do**
    $CouleurUtilisee[i] \leftarrow$ **false**
  end for
  for all $V \in noeud.Voisins$ **tels que** $V.Couleur \neq -1$ **do**
    $CouleurUtilisee[V.Couleur] \leftarrow$ **true**
  end for
  for $i = 0$ **to** Δ **do**
    if $CouleurUtilisee[i] =$ **false then**
      $noeud.Couleur \leftarrow i$
      return
    end if
  end for

ColorierGraphe(*graphe*,Δ)
  for all $N \in$ *graphe* **do**
    ColorierNœud$(noeud, \Delta)$
  end for

Chaque appel de **ColorierNœud** colorie nécessairement le nœud en paramètre puisque, ayant au plus Δ voisins, le tableau *CouleurUtilisee* contient au plus Δ valeurs mise à *true*, et donc au moins une valeur *false*. Ainsi, puisque chaque appel de **ColorierNœud** colorie le nœud donné en paramètre, et qu'il le colorie sans que 2 voisins n'ait jamais la même couleur, à la fin de **ColorierGraphe** le graphe est donc nécessairement $\Delta + 1$-colorié.

L'algorithme parcourant l'ensemble des arrêtes de chaque nœud, et ce pour chaque nœud du graphe, il travaille donc en O(nbr arrête)

5.2.5 Question 5

Le graphe étant 3-coloriable, pour chaque nœud du graphe il est donc possible de colorier tous ses voisins avec 2 couleurs. On sais de plus qu'une fois que tous les nœuds auront moins de $\sqrt{n}$ voisins non coloriés, on pourra "rapidement" colorier le graphe en utilisant l'algorithme de la question 4.

Ainsi, l'algorithme suivant :

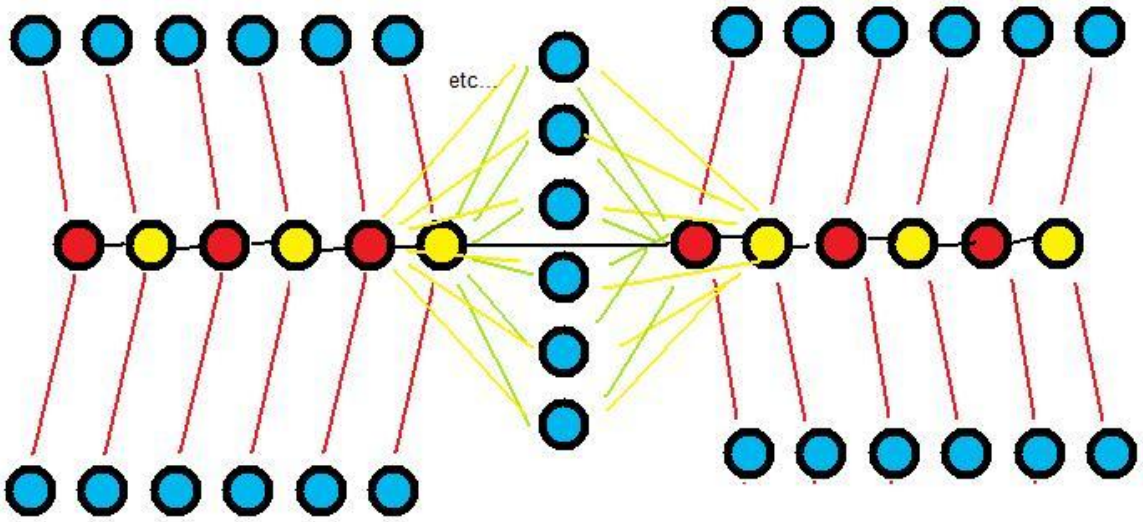
ColorierRapide(*graphe*)
  for all $N \in graphe$ **do**
    $N.Couleur \leftarrow -1$
  end for
  $Couleur \leftarrow \sqrt{n} + 1$ // compte la dernière couleur non utilisée, et on garde les $\sqrt{n} + 1$ premières couleurs pour la seconde partie de l'algorithme
  for all $N \in graphe$ **do**
    $VoisinsNonColoriee \leftarrow 0$
    for all $V \in N.Voisins$ **tels que** $V.Couleur = -1$ **do**
      $VoisinsNonColorie \leftarrow VoisinsNonColorie + 1$
    end for
    if $VoisinsNonColorie > \sqrt{n}$ **then**
      ColorierBinaire$(N.Voisins, Couleur)$;
      $N.Couleur \leftarrow Couleur + 2$
      $Couleur = Couleur + 3$
    end if
  end for
  Colorier$(graphe, \sqrt{n})$

Où **ColorierBilaire** est l'algorithme de la question 3, légèrement modifié pour colorier le graphe en utilisant la couleur donnée en entrée et la suivante, et pour ne colorier que les éléments données dans la liste des nœuds prise en entrée (donc en vérifiant, pour chaque voisin d'un nœud que l'on veux colorier, s'il est bien dans la liste des nœuds donnés en paramètre de la fonction. **Colorier** est exactement l'algorithme de la question 4, en vérifiant que la couleur du nœud est bien strictement inférieur à $\sqrt{n} + 1$ avant de rajouter sa valeur dans le tableau "$CouleurUtilisee$".

Cet algorithme utilise donc au plus 3*$\sqrt{n}$ couleurs dans la première partie (puisqu'il colorie au moins $\sqrt{n}$ nœuds avec 3 couleurs, et qu'il ne peux le faire qu'un maximum de $\sqrt{n}$ fois), puis réutilise au pire $\sqrt{n} + 1$ couleur dans la seconde partie. Il colorie donc bien le graphe en O($\sqrt{n}$) couleurs.

Quand à la complexité, chaque nœud est appelée une fois soit par **ColorierBinaire**, soit par **Colorier**, qui regarde chacun de ses voisins. L'algorithme **ColorierRapide** effectue donc O(k) opérations, où k est le nombre d'arréte du graphe, qui est majoré par n^2, ce qui fait un nombre non exponentiel d'opérations.

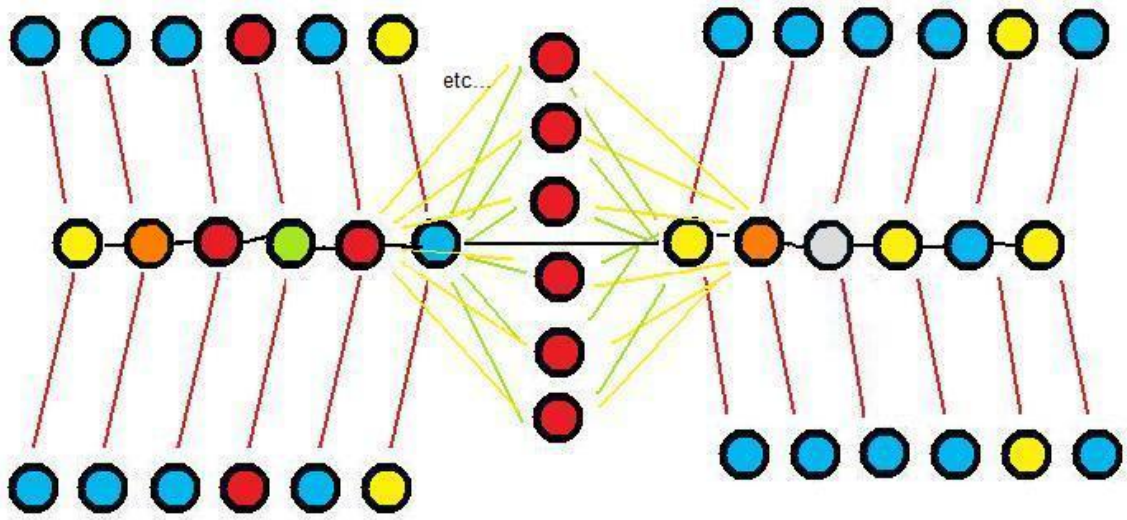
Soit le graphe suivant que l'on veux colorier (qui avait servi à invalider un certains nombre de faux algorithmes en TD...) :



sur lequel il faut comprendre que les $n/4$ nœuds du graphes, placés sur la "verticale" sont tous reliés à l'ensemble des nœuds de la "ligne horizontale centrale" (qui sont donc aussi au nombre de $n/4$). Ce graphe est clairement 3-coloriable avec le coloriage indiqué sur l'image ci-dessus.

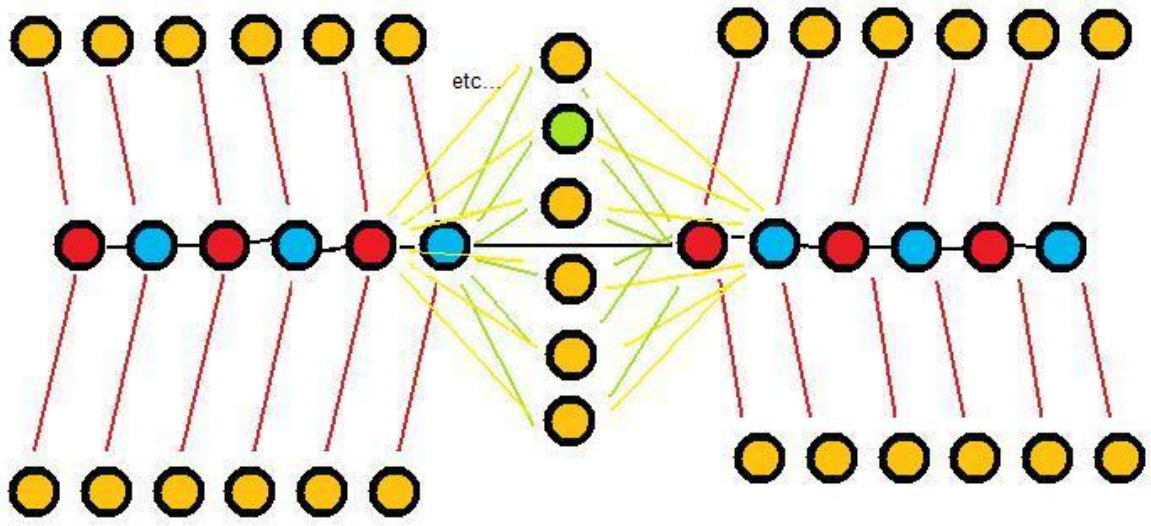
Avec l'algorithme présent, 2 cas de figure sont possible :

- Soit l'algorithme détecte un nœud "horizontal" avant un noeud vertical, auquel cas le graphe sera colorié en 4 à 6 couleur de la façon suivante :



Où le nœud colorié en vert a été sélectionné en premier, et a donc colorié tous ses voisins en rouge. Ensuite l'algorithme de la question 4 a utilisé les couleurs cyan, jaune, orange puis gris pour colorier le reste du graphe, en ayant occasionnellement besoin de la couleur orange et gris s'il n'as "pas eu de chance" lors de la sélection des nœuds (on peux démontrer que ces couleurs suffisent puisque chaque nœud à colorier est reliés à 4 autres nœuds dont 2 sont nécessairement de la même couleur). L'algorithme colorie donc, dans ce cas, rapidement le graphe de façon quasi optimale.

- Sinon, l'algorithme tombe en premier sur un nœud "vertical, et dans ce cas revoit le graphe coloriée de la manière suivante :



où de même le premier nœud rencontré et celui colorié en vert, ses voisins sont colorié en rouge et bleu par l'algorithme de la question 3, et où l'algorithme de la question 4 n'as eu qu'à colorier en orange tous les autres nœuds, puisqu'ils ne sont pas relié entre eux.

5.2.6 Question 6

selon le même principe que la question précédente, pour colorier un graphe k-coloriable, on colorie récursivement avec l'algorithme de $(k-1)$-coloriage tous les nœuds du graphes possédant plus de $O(n^{f(k)}$ voisins, puis on colorie le reste avec l'algorithme de la question 4. On obtiens donc un arbre colorié avec $O(k * n^{f(k)}$ couleurs, à condition que :
$((k-1) * n^{f(k-1)}) * n^{1-f(k)} < k * n^{f(k)}$
puisque l'on veux f(k) minimale, on prendra :
$(k-1/k) * n^{1-f(k)+f(k-1)} = n^{f(k)}$
d'où :
$f(k) = 1/2 * (1 + f(k-1))$
puisque l'on a $f(3) = 1/2$, on obtiens donc la formule : $f(k) = 1 - 1/(2^{k-2})$