Quentin Garnier MI-I5

**Identifier des graphes bipartis**

Mon Algorithme :

L’on doit rajouter l’attribut couleur au sommet, on peut donc faire s.couleur 🡨 bleu par exemple. Le parcours est un simple parcours BFS a la simple différence que nous ne prenons pas que les voisins visités de v, mais tous, afin de vérifier leur coloration pour les comparer avec celle de v. Cependant, nous utilisons toujours l’attribut visited afin de savoir si on peut ajouter le sommet dans la file ou non.

Afin de pouvoir vérifier la bipartition d’un graphe non connexe j’utilise le parcours en largeur qui va me permettre de vérifier la bipartition de chaque composante. Ce dernier est composé de BFS (ici EstBiparti) et de exploreBFS (ici exploreEstBiparti).  
Si il retourne Faux, le graphe n’est pas biparti, dans le cas contraire il retourne vrais

**Procedure** EstBiparti (G)  
 v.visited 🡨 false ∀v ∈ V  
 v.couleur 🡨 blanc ∀v ∈ V //on initialise a une couleur de base  
 **Pour** (v) ∈ V **Faire**  
 **Si** ¬v .visited **Alors**  
 **SI** ¬exploreEstBiparti (G,v) **Alors** //Si la fonction retourne faux  
 **Retourner** FAUX //cette partie du graphe n’est pas bipartie, on retourne faux  
 **FinSI**  
 **FinSi**  
 **FinPour  
 Retourner** Vrais //Si on sort de la boucle sans un seul faux , c’est que le graphe est biparti, on retourne vrais **FinProcedure**

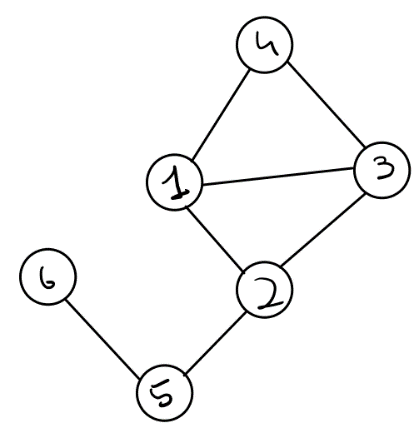
**Procedure** ChangeColor (couleur) //Permet d’inter changer les couleurs entre chaque niveau de noeud  
 **Si** couleur = bleu **Alors**  
 **retourner** rouge  
 **Sinon**  
 **retourner** bleu  
 **FinSi**  
**FinProcedure**

**Procedure** exploreEstBiparti(G,s)  
 s.visited 🡨 true  
 s.couleur 🡨 rouge //on commence avec la couleur rouge arbitrairement  
 open.add(s)  
 **TantQue** ¬open.empty() **Faire** v 🡨 open.peek() **Pour** (u) = chaque voisin de v **Faire**   
 u 🡨 un des voisin de v  
 **Si** u existe **Alors  
 Si** u.couleur = v.couleur **Alors** //Si il a un voisin de la même couleur que lui **retourner** FAUX //le graphe contient un cycle impaire, et n’est pas biparti  
 **Sinon  
 SI** ¬u.visited **Alors** u.visited 🡨true  
 u.couleur🡨ChangeColor(v.couleur) //on donne la couleur inverse au voisin  
 Open.add(u)  
 **FinSi**  
 **FinSi**   
 **Sinon** //Si u n’existe pas, on a fini de parcourir les sommet voisin de v, on le retire donc  
 Open.remove()   
  **FinSi  
 FinPour  
 FinTantQue  
FinProcedure**

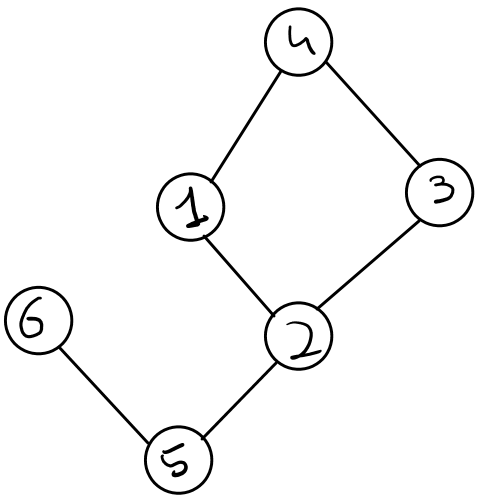
Explication du fonctionnement de mon algorithme :

* Comment se fait-il qu’il réponde que le graphe n'est pas biparti quand il ne l'est pas ?  
  **Si** u.couleur = v.couleur **Alors   
  retourner** FAUX  
  Cette simple condition constitue la condition d’arrêt du programme. Située dans exploreEstBiparti, elle s’exécute pour chaque voisin de v. Mais, qu’est-ce que cela signifie si u et v sont de la même couleur ? Simplement, un graphe biparti et composé de seulement deux stables. Chaque sommet de ses deux stables sont de la même couleur (ici, un stable rouge, et un autre bleu). Parcourir un graphe en largeur en alternant les couleur entre le sommet parent et son enfant nous permet de créer ces fameux deux stables. A travers ExploreEstBiparti, nous vérifions donc que la création de deux stable est possible. Nous constatons ainsi, que cette dernière n’est pas possible à partir du moment où nous avons deux sommet voisin qui sont de la même couleur (ce qui n’est pas possible dans un stable). En effet, cela témoigne de la présence d’un cycle impaire, ce qui est caractéristique d’un graphe non Biparti. Une fois que ExploreEstBiparti a retourné FAUX, EstBiparti récupère ce FAUX et retourne à son tour FAUX, indiquant que le graphe n’est pas biparti.
* Comment se fait-il qu’il réponde que le graphe est biparti quand il l’est ?  
  Simplement, une fois que la file est vide, si l’algorithme n’a pas retourné faux, c’est qu’il n’a pas trouvé de voisin de même couleur, et que donc le graphe a pu être découpé en deux stable (un rouge et un bleu). Open est donc vide, l’algorithme sort de la boucle et retourne VRAIS. Si pour chaque composante connexe ExploreEstBiparti retourne VRAIS, alors, la condition d’arrêt de EstBiparti qui retourne FAUX n’est jamais remplie, on sort donc de la boucle ( **Pour** (v) ∈ V **Faire** ) et on retourne ainsi VRAIS témoignant du fait que le graphe est biparti.

Exécution de l’algorithme :

* Sur un graphe non Biparti :

|  |  |  |
| --- | --- | --- |
| Graphe | instruction | open |
|  | u.couleur 🡨Blanc pour tout u  u.visited 🡨false pour tout u  1.visited 🡨 true 1.couleur 🡨 rouge  open .add [1] | [1] |
|  | v = 1 et u = 2  2 existe && 2.couleur différent 1.couleur && 2.visited = false **DONC :**  2.visited 🡨true 2.couleur🡨ChangeColor(1.couleur)  Open.add(2) | [1,2] |
|  | v = 1 et u = 3  3 existe && 3.couleur différent 1.couleur && 3.visited = false **DONC :**  3.visited 🡨true 3.couleur🡨ChangeColor(1.couleur)  Open.add(3) | [1,2,3] |
|  | v = 1 et u = 4  4 existe && 4.couleur différent 1.couleur && 4.visited = false **DONC :**  4.visited 🡨true 4.couleur🡨ChangeColor(1.couleur)  Open.add(4) | [1,2,3,4] |
|  | Il n’y a plus de voisin de 1 **DONC** : open.remove() | [2,3,4] |
|  | v = 2 et u = 5  5 existe && 5.couleur différent 2.couleur && 5.visited = false **DONC :**  5.visited 🡨true 5.couleur🡨ChangeColor(2.couleur)  Open.add(5) | [2,3,4,5] |
|  | v = 2 et u = 3  3 existe && 3.couleur = 2.couleur  **DONC :**  exploreEstBiparti return False  **DONC :**  EstBiparti return False | [2,3,4,5] |

* Sur un graphe biparti :

|  |  |  |
| --- | --- | --- |
| Graphe | instruction | open |
|  | u.couleur 🡨Blanc pour tout u  u.visited 🡨false pour tout u  1.visited 🡨 true 1.couleur 🡨 rouge  open .add [1] | [1] |
|  | v = 1 et u = 2  2 existe && 2.couleur différent 1.couleur && 2.visited = false **DONC :**  2.visited 🡨true 2.couleur🡨ChangeColor(1.couleur)  Open.add(2) | [1,2] |
|  | v = 1 et u = 4  4 existe && 4.couleur différent 1.couleur && 4.visited = false **DONC :**  4.visited 🡨true 4.couleur🡨ChangeColor(1.couleur)  Open.add(4) | [1,2,4] |
|  | Il n’y a plus de voisin de 1 **DONC** : open.remove() | [2,4] |
|  | v = 2 et u = 3  3 existe && 3.couleur différent 2.couleur && 3.visited = false **DONC :**  3.visited 🡨true 3.couleur🡨ChangeColor(2.couleur)  Open.add(3) | [2,4,3] |
|  | v = 2 et u = 5  5 existe && 5.couleur différent 2.couleur && 5.visited = false **DONC :**  5.visited 🡨true 5.couleur🡨ChangeColor(2.couleur)  Open.add(5) | [2,4,3,5] |
|  | v = 2 et u = 1  1 existe && 1.couleur différent 2.couleur && 1.visited = true **DONC :**  On ne fait rien | [2,4,3,5] |
|  | Il n’y a plus de voisin de 2 **DONC** : open.remove() | [4,3,5] |
|  | (je fais les deux voisin de 4 même temps car il remplissent les même conditions)  v = 2 et u = 1 et 3  1 et 3 existe && 1.couleur et 3.couleur différent 2.couleur && 1.visited et 3.visited = true **DONC :**  On ne fait rien | [4,3,5] |
|  | Il n’y a plus de voisin de 4 **DONC** : open.remove() | [3,5] |
|  | (je fais les deux voisin de 3 même temps car il remplissent les même conditions)  v = 2 et u = 2 et 4  2 et 4 existe && 2.couleur et 4.couleur différent 3.couleur && 2.visited et 4.visited = true **DONC :**  On ne fait rien | [3,5] |
|  | Il n’y a plus de voisin de 3 **DONC** : open.remove() | [5] |
|  | v = 5 et u = 2  2 existe && 2.couleur différent 5.couleur && 2.visited = true **DONC :**  On ne fait rien | [5] |
|  | v = 5 et u = 6  6 existe && 6.couleur différent 5.couleur && 6.visited = false **DONC :**  6.visited 🡨true 6.couleur🡨ChangeColor(5.couleur)  Open.add(6) | [5,6] |
|  | Il n’y a plus de voisin de 5 **DONC** : open.remove() | [6] |
|  | v = 6 et u = 5  5 existe && 5.couleur différent 6.couleur && 5.visited = true **DONC :**  On ne fait rien | [6] |
|  | Il n’y a plus de voisin de 6 **DONC** : open.remove() | [] |
|  | Open.empty()=true  **DONC :**  On sort de la boucle  **DONC :**  exploreEstBiparti return False  **DONC :**  EstBiparti return False | [] |