Graphes bipartis et couplage

OPTION INFORMATIQUE - TP n° 3.5 - Olivier Reynet

À la fin de ce chapitre, je sais :

- ☞ reconnaître et caractériser un graphe biparti
- ☞ utiliser le parcours en largeur d'un graphe pour caractériser un graphe biparti
- ☞ établir un chemin alternant
- ☞ trouver un couplage de cardinal maximum

A Graphes bipartis ou bi-colorables

■ **Définition 1 — Graphe biparti.** un graphe $G = (V, E)$ est biparti si l'ensemble V de ses sommets peut être divisé en deux sous-ensembles disjoints U et W tels que chaque arête de E possède une extrémité dans U et l'autre dans W.

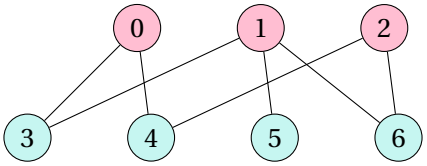


FIGURE 1 – Graphe biparti

Théorème 1 — Caractérisation des graphes bipartis. Un graphe est biparti si et seulement s'il ne possède aucun cycle de longueur impair.

Pour savoir si un graphe est biparti on pourrait donc rechercher les cycles et évaluer leurs longueurs. Il existe cependant une autre approche. Un graphe biparti est un graphe bicolorable comme le montre la figure 1.

B Coloration et bi-coloration de graphes

■ **Définition 2 — Coloration.** Une coloration d'un graphe simple est l'attribution d'une couleur aux sommets de ce graphe.

■ **Définition 3 — Coloration valide.** Une coloration est valide lorsque deux sommets adjacents n'ont jamais la même couleur.

■ **Définition 4 — Nombre chromatique.** Le nombre chromatique d'un graphe G est le plus petit nombre de couleurs nécessaires pour obtenir une coloration valide de ce graphe. On le note généralement $\chi(G)$.

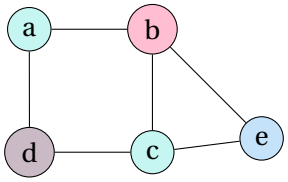


FIGURE 2 – Exemple de 4-coloration valide d'un graphe. Cette coloration n'est pas optimale.

■ **Définition 5 — k-coloration.** Lorsqu'une coloration de graphe utilise k couleurs, on dit d'elle que c'est une k-coloration.

■ **Définition 6 — Coloration optimale.** Une $\chi(G)$-coloration valide est une coloration optimale d'un graphe G.

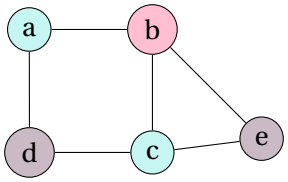


FIGURE 3 – Exemple de 3-coloration valideă d'un graphe. Cette coloration est optimale.

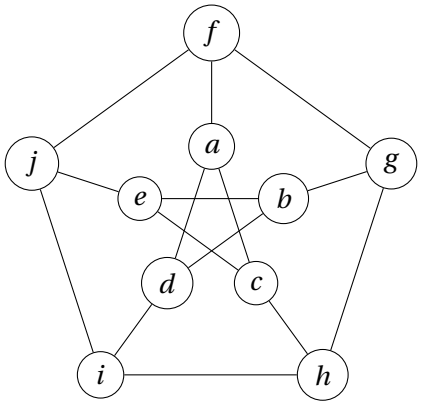


FIGURE 4 – Graphe de Petersen : saurez-vous proposer une coloration optimale de ce graphe sachant que son nombre chromatique vaut trois ?

C Algorithme de test de la bi-colorabilité d'un graphe

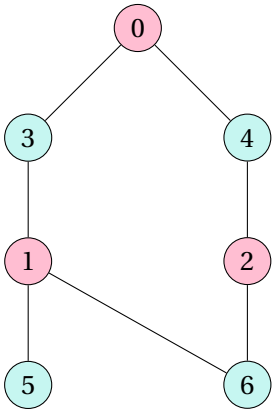
Pour tester la bi-colorabilité d'un graphe, il suffit de parcourir le graphe en largeur et de colorer les sommets d'un même niveau de la même couleur. L'algorithme s'arrête si deux sommets adjacents possèdent la même couleur.

C1. Montrer que si un graphe est biparti alors il est au pire bicolorable.

Solution : Soit $G = (U, W, A)$ un graphe biparti. S'il ne posède aucune arête, il est 1-colorable. S'il possède des arêtes, alors choisissons le rouge pour les sommets de U et le bleu pour les sommets de W. Cette coloration est valide car aucun sommet adjacent ne possède la même couleur.

C2. Effectuer à la main un parcours en largeur du graphe de la figure 1 en partant du sommet 0. En déduire un dessin du graphe faisant apparaître quatre niveaux. Ce graphe est-il bi-colorable ?

Solution : Le parcours en largeur à partir de 0 donne `[0; 3; 4; 1; 2; 5; 6]`. On peut donc redessiner le graphe comme suit pour faire apparaître quatre niveaux :



Ce graphe est clairement bi-colorable !

C3. Créer un type somme `color_type` sont les valeurs sont `White`, `Black` ou `Orange`.

Solution :

```
type color_type = White | Black | Orange;;
```

C4. Créer trois variables globales `cw`, `cb` et `co` de type `color_type` dont les valeurs sont respectivement `White`, `Black` et `Orange`.

Solution :

```
let co = Orange;;
let cb = Black;;
let cw = White;;
```

C5. Créer une fonction de signature `color_to_string : color_type -> string` qui transforme un type `color_type` en chaîne de caractère en utilisant le pattern matching. Par exemple, `White` devient `"White"`.

Solution :

```
let color_to_string color =  match color with
  | White -> "White"
  | Black -> "Black"
  | Orange -> "Orange";;
```

C6. Créer une fonction de signature `next_color : color_type -> color_type` qui renvoie la couleur du prochain niveau en utilisant le pattern matching. Si le paramètre d'entrée est `Black` elle renvoie `Orange` et inversement. Si le paramètre d'entrée est `White`, elle échoue en renvoyant le message `"Can not color in white !"` : on ne peut pas colorer en blanc dans l'algorithme, c'est juste la couleur initiale des sommets non explorés.

Solution :

```
let next_color c = match c with
    | Black -> Orange
    | Orange -> Black
    | White -> failwith "Can not color in white !";;
```

C7. Créer un exception `Not2colorable`.

Solution :

```
exception Not2colorable;;
```

C8. Créer une fonction récursive de signature `color_neighbours : 'a list -> 'b -> ('a * 'b)list` qui prend comme paramètre une liste de sommets et une couleur et qui renvoie la liste des tuples de chaque sommet adjoint à la couleur. Par exemple, avec les paramètres `[1;2;4]` et `Blue`, la fonction renvoie `[(1,Blue);(2,Blue);(4,Blue)]`.

Solution :

```
let rec color_neighbours neighbours c = match neighbours with
    | [] -> []
    | v::t -> (v,c)::(color_neighbours t c);;
```

C9. Coder un algorithme de parcours en largeur afin de statuer sur la bi-colorabilité d'un graphe. Cette implémentation récursive renvoie `false` si l'exception `Not2colorable` est levée. Elle renvoie `true` sinon. On remarquera qu'on peut parcourir en largeur un graphe sans pour autant renvoyer la liste des sommets parcourus : dans le cas présent, on n'en a pas besoin. Enfin, la fonction `color_neighbours`

est utile pour construire une file de voisins colorés à l'identique. La fonction `next_color` est utile pour choisir la couleur du niveau suivant. La fonction `color_to_string` est utile pour afficher sur la console l'évolution de l'algorithme. Suivre le squelette de code suivant :

```
let bicolorable g v0 =
  let color = Array.make (Array.length g) White in
    let rec explore queue = (* queue -> file en anglais FIFO *)
      match queue with
        | ... (* End case, we don't need the path list so return unit () *)
        | ... (* color vertex and neighbours and keep on exploring *)
        | ... (* already colored OK, exploring remaining vertices *)
        | ... (* stop exploring, raise execption Not2colorable  *)
  in try explore [(v0,Black)]; true with (* return true * )
         | Not2colorable -> false;; (* catching exception, return false *)
```

C10. Tester cette implémentation sur le graphe de la figure 1 `let bg = [| [3;4] ; [3;5;6] ; [4;6] ; [0;1] ; [0;2] ; [1] ; [1;2] |]`.

C11. Tester l'implémentation sur le graphe `let bg_mod = [| [2;3;4] ; [3;5;6] ; [4;6;0] ; [0;1] ; [0;2] ; [1] ; [1;2] |]`.

Solution :

```
let bg = [| [3;4] ; [3;5;6] ; [4;6] ; [0;1] ; [0;2] ; [1] ; [1;2] |] ;;
let bg_mod = [| [2;3;4] ; [3;5;6] ; [4;6;0] ; [0;1] ; [0;2] ; [1] ; [1;2] |]
     ;;

let bicolorable g v0 =
  let color = Array.make (Array.length g) White in
  let rec explore queue = (* queue -> file en anglais FIFO *)
    match queue with
      | [] -> ()  (* we dont need he path list --> unit () *)
      | (v,c)::t when color.(v) = White ->
        color.(v) <- c;
        Printf.printf "%i from white to %s\n" v (color_to_string c);
        explore (t @ (color_neighbours g.(v) (next_color c)))
      | (v,c)::t when color.(v) = c ->
        Printf.printf "Vertex %i will remain %s\n" v (color_to_string c);
        explore t    (* keep on exploring *)
      | (_,_)::_ -> raise Not2colorable  (* stop exploring *)
  in try explore [(v0,Blue)]; true with
      |  Not2colorable -> false;;

bicolorable bg 0 ;;
bicolorable bg_mod 0 ;;
```