Coloration de graphe - Correction

Samy Jaziri

1 Mise en jambe

Soit G un graphe dont le degré maximal des sommets est Δ.
L'utilisation d'une couleur supplémentaire dans un algorithme de coloriage n'est nécessaire que dans le cas où, lors du coloriage d'un sommet, l'ensemble des couleurs de ses voisins regroupe déjà toutes les couleurs utilisée. Or un sommet a au plus Δ voisins. Si nous partons d'une palette de $\Delta + 1$ couleurs, il sera toujours possible lors du coloriage d'un sommet de trouver une couleur de la palette qui n'est pas utilisée par ses voisins (puisqu'il n'y a que Δ voisins).
Pour une clique à n sommets, il est évident que le seul coloriage possible est un n coloriage. Le degré maximal d'un sommet est aussi le degré de chaque sommet et est égal à $n - 1$. La borne est donc atteinte. Cette observation permet de répondre à la question suivante. La clique à $k + 1$ sommet n'est pas k-coloriable.

```
(* Les graphes seront représentés sous forme de listes d'adjacence *)
type sommet = int ;;
type graphe = ( sommet list ) array ;;
type couleur = AUCUNE | COULEUR of int ;;
```

```
let est_2_coloriable ( g : graphe ) : bool =
  let inverse = function
      | AUCUNE -> AUCUNE
      | COULEUR(i) -> COULEUR( (i + 1) mod 2 ) in
  let couleur = Array.make ( Array.length g ) AUCUNE in
  let a_visiter = Queue.create () in
  let rec parcours_composante ( a_visiter : sommet Queue.t ) =
    if Queue.is_empty a_visiter then true
    else
    begin
      let s = Queue.pop a_visiter in
      let rec parcours_voisins ( voisins : sommet list ) : bool =
        match voisins with
        | [] -> true
        | v :: voisins when couleur.(v) = couleur.(s) -> false
        | v :: voisins when couleur.(v) = AUCUNE ->
          begin
            couleur.(v) <- (inverse couleur.(s)) ;
            Queue.push v a_visiter ;
            parcours_voisins voisins
          end
        | v :: voisins -> parcours_voisins voisins in
      if parcours_voisins g.(s) then parcours_composante a_visiter
      else false
    end in
  let rec parcours ( i : sommet ) : bool =
    if i = Array.length g then true
    else
      if couleur.(i) = AUCUNE then
      begin
        couleur.(i) <- COULEUR(0);
        Queue.push i a_visiter;
        if parcours_composante a_visiter then parcours ( i + 1 )
        else false
      end
      else parcours ( i + 1 ) in
  parcours 0
;;
```

Supposons qu'un graphe est bipartie. On peut séparer les sommets de G en deux sous-ensemble R et N tels que les voisins de R soient tous dans N et inversement. On colorie les éléments de R en *rouge* et les éléments de N en *noir*. Par définition nous avons bien une 2-coloration de G.
Inversement si G est deux coloriable. Soit R les sommets colorés en *rouge* et N ceux colorés en *noir*. Par définitions les voisins de R ne sont pas dans R et sont donc dans N, et inversement. G est donc bipartie.

2 Approximation du nombre chromatique

L'algorithme glouton consiste en un parcours en largeur :

- le premier sommet visité est coloré en 0
- On colorie chaque sommet visité avec la couleur k minimale telle que aucun voisin du sommet n'est coloré avec la couleur k.

Dans le graphe G_1, les noeuds sont numéroté par leur ordre de visite dans un parcours en largeur. On ordonne les couleurs comme ceci : blue, rose, vert, jaune. L'algorithme glouton décrit ci-dessus choisi de colorier le troisième sommet avec la couleur verte qui est la plus petite non utilisée par un voisin de ce sommet. Ce qui force ensuite le sommet 4 à utiliser une quatrième couleur. Pourtant, en coloriant le sommet 3 avec la couleur rose, on aurait pu utiliser la couleur verte pour le sommet 4.

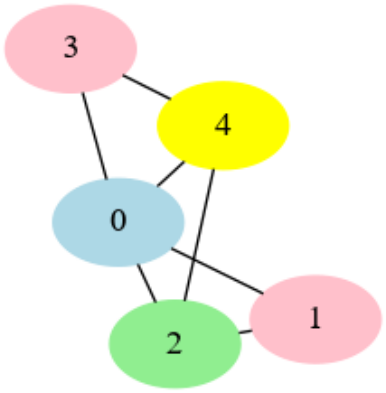

Figure 1: Graphe G_1 colorié par l'algorithme.

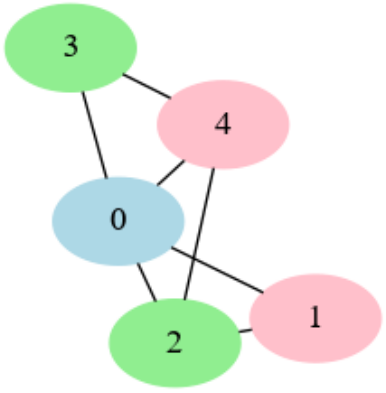

Figure 2: Graphe G_1 3-coloriable

```
let coloration_glouton ( g : graphe ) : couleur array =
  let couleur = Array.make ( Array.length g ) AUCUNE in
  let visite_prevue = Array.make ( Array.length g ) false in
  let a_visiter = Queue.create () in
  let rec parcours_composante
            ( a_visiter : sommet Queue.t ) ( nb_couleur : int ) : unit =
    if Queue.is_empty a_visiter then ()
    else
    begin
      let couleur_utilisee = Array.make nb_couleur false in
      let s = Queue.pop a_visiter in
      let rec parcours_voisins ( voisins : sommet list ) : unit =
        match voisins with
        | [] -> ()
        | v :: voisins -> match couleur.(v) with
          | AUCUNE -> ( if not visite_prevue.(v) then
                          ( Queue.push v a_visiter ; visite_prevue.(v) <- true );
                            parcours_voisins voisins )
          | COULEUR(k) -> ( couleur_utilisee.(k) <- true;
                            parcours_voisins voisins ) in
      parcours_voisins g.(s);
      let couleur_disponible = ref None in
      let i = ref 0 in
      while !couleur_disponible = None && !i < nb_couleur do
        if not (couleur_utilisee.(!i)) then
          couleur_disponible := Some(!i)
        else incr i
      done;
      match !couleur_disponible with
      | None -> ( couleur.(s) <- COULEUR(nb_couleur);
                  parcours_composante a_visiter ( nb_couleur + 1 ) )
      | Some(x) -> ( couleur.(s) <- COULEUR(x) ;
                     parcours_composante a_visiter nb_couleur )
    end in
  let rec parcours ( i : sommet ) : unit =
    if i = Array.length g then ()
    else
    begin
      if couleur.(i) = AUCUNE then
      begin
        Queue.push i a_visiter;
        visite_prevue.(i) <- true;
        parcours_composante a_visiter 1;
      end;
      parcours ( i + 1 )
    end in
  parcours 0;
  couleur
;;
```

Cet algorithme de coloration glouton est en $\mathcal{O}(|V|\,|\Delta|)$ si on borne de le nombre de couleurs par $\Delta+1$ le degré maximal d'un sommet.
Pour l'algorithme de Welsh et Powell on trie les sommets par ordre de degré, puis on utilise l'heuristique gloutonne précédente sur chaque sommet dans l'ordre.

```
let coloration_welsh_powell ( g : graphe ) : couleur array =
  let nb_sommet = Array.length g in
  let couleur = Array.make nb_sommet AUCUNE in
  let degre = Array.make nb_sommet 0 in
  let tri_sommet = Array.make nb_sommet 0 in
  let nb_couleur = ref 1 in
  for i = 0 to nb_sommet - 1 do
    degre.(i) <- List.length g.(i) ;
    tri_sommet.(i) <- i
  done;
  Array.stable_sort (fun x y -> compare degre.(x) degre.(y)) tri_sommet;
  for i = 0 to nb_sommet - 1 do
    let couleur_utilisee = Array.make !nb_couleur false in
    let s = tri_sommet.(i) in
    let rec parcours_voisins ( voisins : sommet list ) : unit =
      match voisins with
      | [] -> ()
      | v :: voisins -> match couleur.(v) with
        | AUCUNE -> parcours_voisins voisins
        | COULEUR(k) -> ( couleur_utilisee.(k) <- true;
                          parcours_voisins voisins ) in
    parcours_voisins g.(s);
    let couleur_disponible = ref None in
    let i = ref 0 in
    while !couleur_disponible = None && !i < !nb_couleur do
      if not (couleur_utilisee.(!i)) then
        couleur_disponible := Some(!i)
      else incr i
    done;
    match !couleur_disponible with
    | None -> ( couleur.(s) <- COULEUR(!nb_couleur) ; incr nb_couleur )
    | Some(x) -> couleur.(s) <- COULEUR(x) ;
    parcours_voisins g.(s);
  done ;
  couleur
;;
```

Le calcul des degré est en $O(|E|)$. Le tri est en $\mathcal{O}(|V|\log(|V|))$. Le coloriage séquentiel est en $O(\Delta\,|V|)$. Le reste des opérations est en $\mathcal{O}(|V|)$. La complexité finale est $\mathcal{O}(|V|\,(\Delta+\log(|V|)))$ comme $|E|=\mathcal{O}(\Delta\,|V|)$.
Les couronnes à $2n$ éléments sont 2-coloriable mais selon l'ordre de parcours de l'algorithme on peut aller jusqu'à une coloration à n couleurs. Voir l'exemple pour $n=4$.

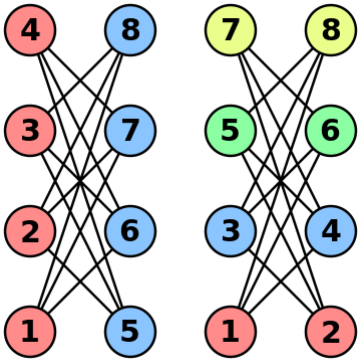

Figure 3: Graphe couronne à 8 éléments (wikipedia)

3 Calcul exact du nombre chromatique

```
let est_3coloriable ( g : graphe ) : bool =
  let nb_sommet = Array.length g in
  let visite_prevue = Array.make nb_sommet false in
  let rec parcours_composante
            ( a_visiter : sommet Queue.t )
            ( couleur : couleur array ) : bool =
    if Queue.is_empty a_visiter then true
    else
    begin
      let couleur_utilisee = Array.make 3 false in
      let s = Queue.pop a_visiter in
      visite_prevue.(s) <- true;
      let rec parcours_voisins ( voisins : sommet list ) : unit =
        match voisins with
        | [] -> ()
        | v :: voisins -> match couleur.(v) with
          | AUCUNE -> ( if not visite_prevue.(v) then
                          ( Queue.push v a_visiter ; visite_prevue.(v) <- true );
                            parcours_voisins voisins )
          | COULEUR(k) -> ( couleur_utilisee.(k) <- true;
                            parcours_voisins voisins ) in
      parcours_voisins g.(s);
      let est_3coloriable = ref false in
      let couleur_testee = ref 0 in
      while !couleur_testee < 3 && not !est_3coloriable do
        if not couleur_utilisee.(!couleur_testee) then
        begin
          let copy_couleur = Array.copy couleur in
          let copy_a_visiter = Queue.copy a_visiter in
          copy_couleur.(s) <- COULEUR(!couleur_testee);
          est_3coloriable := parcours_composante copy_a_visiter copy_couleur;
        end;
        incr couleur_testee
      done;
      !est_3coloriable
    end in
```

```
  let rec parcours ( i : sommet ) : bool =
    if i = Array.length g then true
    else
    begin
      if not visite_prevue.(i) then
      begin
        let a_visiter = Queue.create () in
        Queue.push i a_visiter;
        visite_prevue.(i) <- true;
        if parcours_composante a_visiter (Array.make nb_sommet AUCUNE)
        then parcours ( i + 1 )
        else false
      end
      else parcours ( i + 1 )
    end in
  parcours 0;
;;
```

```
let nombre_chromatique ( g : graphe ) : int =
  let nb_sommet = Array.length g in
  let visite_prevue = Array.make nb_sommet false in
  let rec parcours_composante
            ( a_visiter : sommet Queue.t )
            ( nb_couleur : int )
            ( couleur : couleur array ) : int =
    if Queue.is_empty a_visiter then nb_couleur
    else
    begin
      let couleur_utilisee = Array.make nb_couleur false in
      let s = Queue.pop a_visiter in
      let rec parcours_voisins ( voisins : sommet list ) : unit =
        match voisins with
        | [] -> ()
        | v :: voisins -> match couleur.(v) with
          | AUCUNE -> ( if not visite_prevue.(v) then
                          ( Queue.push v a_visiter ; visite_prevue.(v) <- true );
                            parcours_voisins voisins )
          | COULEUR(k) -> ( couleur_utilisee.(k) <- true;
                            parcours_voisins voisins ) in
      parcours_voisins g.(s);
      let nb_chromatique = ref ( (Array.length g) + 1 ) in
      for i = 0 to nb_couleur - 1 do
        if not couleur_utilisee.(i) then
        begin
          let copy_couleur = Array.copy couleur in
          let copy_a_visiter = Queue.copy a_visiter in
          copy_couleur.(s) <- COULEUR(i);
          nb_chromatique :=
            min
              !nb_chromatique
              ( parcours_composante copy_a_visiter nb_couleur copy_couleur ) ;
        end
      done;
      let copy_couleur = Array.copy couleur in
      let copy_a_visiter = Queue.copy a_visiter in
      copy_couleur.(s) <- COULEUR(nb_couleur);
      nb_chromatique :=
        min
          !nb_chromatique
          ( parcours_composante
              copy_a_visiter ( nb_couleur + 1 ) copy_couleur ) ;
        !nb_chromatique
    end in
```

```
  let rec parcours ( i : sommet ) ( nb_chromatique : int ): int =
    if i = Array.length g then nb_chromatique
    else
    begin
      if not visite_prevue.(i) then
      begin
        let a_visiter = Queue.create () in
        Queue.push i a_visiter;
        visite_prevue.(i) <- true;
        let nb_chromatique_composante =
          parcours_composante a_visiter 1 (Array.make nb_sommet AUCUNE) in
        parcours ( i + 1 ) ( max nb_chromatique nb_chromatique_composante )
      end
      else
        parcours ( i + 1 ) nb_chromatique
    end in
  parcours 0 1;
;;
```

4 Affichage des résultats de la fiche réponse

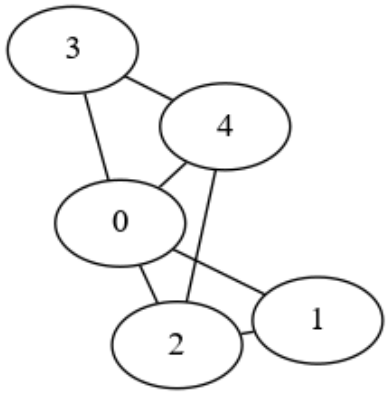

Figure 4: g1 - 3 coloriable.

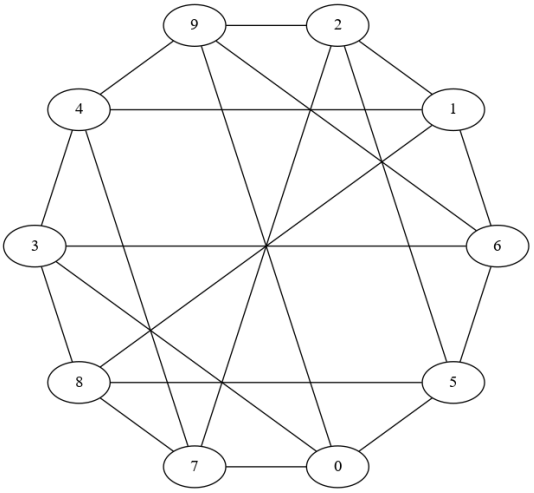

Figure 5: g2 - couronne à 10 sommets

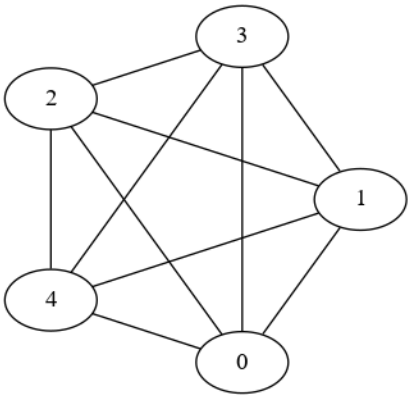

Figure 6: g3 - clique à 5 sommets

```
let g1 : graphe =
  [|
    (* 0 *) [1 ; 2 ; 3 ; 4 ] ;
    (* 1 *) [0 ; 2 ] ;
    (* 2 *) [0 ; 1 ; 4] ;
    (* 3 *) [0 ; 4] ;
    (* 4 *) [0 ; 2 ; 3 ] ;
  |];;

let g2 : graphe =
  [|
    (* 0 *) [3 ; 5 ; 7 ; 9] ;
    (* 1 *) [2 ; 4 ; 6 ; 8] ;
    (* 2 *) [1 ; 5 ; 7 ; 9] ;
    (* 3 *) [0 ; 4 ; 6 ; 8] ;
    (* 4 *) [1 ; 3 ; 7 ; 9] ;
    (* 5 *) [0 ; 2 ; 6 ; 8] ;
    (* 6 *) [3 ; 5 ; 1 ; 9] ;
    (* 7 *) [0 ; 2 ; 4 ; 8] ;
    (* 8 *) [1 ; 5 ; 7 ; 3] ;
    (* 9 *) [0 ; 2 ; 4 ; 6] ;
  |];;

let g3 : graphe =
  [|
    (* 0 *) [1 ; 2 ; 3 ; 4] ;
    (* 1 *) [0 ; 2 ; 3 ; 4] ;
    (* 2 *) [0 ; 1 ; 3 ; 4] ;
    (* 3 *) [0 ; 1 ; 2 ; 4] ;
    (* 4 *) [0 ; 1 ; 2 ; 3] ;
  |];;
```

```
let s_couleur ( c : couleur array ) : string =
  String.concat "--"
    ( List.map
        ( fun c -> match c with
            | AUCUNE -> "-1"
            | COULEUR(k) -> string_of_int k )
        ( Array.to_list c ) ) ;;

Printf.printf "Question 1 : \n|- %s\n"
  ( String.concat "\n|- "
      ( List.map
          ( fun g -> if ( est_2_coloriable g ) then "Oui" else "Non" )
          [g1 ; g2 ; g3  ] ) ) ;;

Printf.printf "Question 2 : \n|- %s\n"
  ( String.concat "\n|- "
      ( List.map
          ( fun g -> s_couleur ( coloration_glouton g ) )
          [g1 ; g2 ; g3  ] ) ) ;;

Printf.printf "Question 3 : \n|- %s\n"
  ( String.concat "\n|- "
      ( List.map
          ( fun g -> s_couleur ( coloration_welsh_powell g ) )
          [g1 ; g2 ; g3  ] ) ) ;;

Printf.printf "Question 4 : \n|- %s\n"
  ( String.concat "\n|- "
      ( List.map
          ( fun g -> if ( est_3coloriable g ) then "Oui" else "Non" )
          [g1 ; g2 ; g3  ] ) ) ;;

Printf.printf "Question 4 : \n|- %s\n"
  ( String.concat "\n|- "
      ( List.map
          ( fun g -> string_of_int ( nombre_chromatique g ) )
          [g1 ; g2 ; g3 ] ) ) ;;
```