ENSTA Paris
2023-2024

Résolution de problèmes algorithmiques – IN103

Alexandre Chapoutot

Feuille d'exercices 5

Objectif(s)

- Manipuler des graphes ;
- Écrire un parcours de graphes pour résoudre des problèmes comme trouver la sortie d'un labyrinthe ou calculer l'existence d'un chemin ;
- Introduction au problème de la coloration de graphes.

PRÉPARATION

Cette première partie permet de préparer votre environnement (**si vous ne l'avez pas fait au dernier TD**) de travail afin de pouvoir utiliser facilement la bibliothèque logicielle **libin103** spécialement développée pour cet enseignement.

1. A la racine de votre compte, créez un répertoire nommé `Library` s'il n'a pas déjà été créé, puis placez vous dans ce répertoire.

   ```
   mkdir ~/Library; cd ~/Library
   ```

2. Téléchargez l'archive `libin103-1.4.tar.gz` sur le site du cours

   ```
   wget https://perso.ensta-paris.fr/~chapoutot/teaching/in103/practical-work/libin103-1.4.tar.gz
   ```

3. Désarchivez l'archive

   ```
   tar -xvzf libin103-1.4.tar.gz
   ```

4. Allez dans le répertoire `libin103-1.4` et compilez la bibliothèque. Quelle commande faut-il utiliser ?
   — Il faut utiliser la commande `make`. A la fin de la compilation vérifier la présence du fichier `libin103.a` dans le répertoire `source`.
   — Également, vous pouvez exécuter la commande `make check` pour compiler et exécuter les programmes de tests.

Matériel pour le TP

Récupérez l'archive associé à cette séance de TP à l'adresse :

```
https://perso.ensta-paris.fr/~chapoutot/teaching/in103/practical-work/in103-td5.tar.gz
```

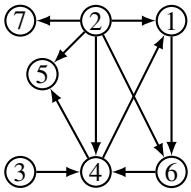
EXERCICES

Exercice 1 – Représentations mathématico-informatiques des graphes

Fonctions utiles pour cet exercice

- `integer_graph_init`
- `integer_graph_destroy`
- `integer_graph_ins_vertex`
- `integer_graph_ins_edge`

Nous allons considérer le graphe suivant



Question 1

Donnez à la main la description du graphe sous la forme d'une matrice d'adjacence.

Solution:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Question 2

Donnez à la main la description du graphe sous la forme d'une liste d'adjacence.

Solution:
On utilise une représentation simple (pas nécessaire de dessiner les structures de listes)

- 1 : 6
- 2 : 1, 4, 5, 6, 7
- 3 : 4
- 4 : 1, 5
- 5 : -
- 6 : 4
- 7 : -

Question 3

À l'aide du programme donné en cours et du squelette de programme donné dans le répertoire `exo1`, construisez informatiquement le graphe à l'aide de la bibliothèque `libin103` et affichez le pour vérifier que vous obtenez une liste d'adjacence équivalente à celle construite à la main.

Solution:

```
int main (int argc, char** argv) {
  integer_graph_t graph;
  integer_graph_init (&graph);

  for (int i = 1; i <= 7 ; i++) {
    integer_graph_ins_vertex(&graph, i);
  }
  integer_graph_ins_edge(&graph, 1, 6, 0.0);
  integer_graph_ins_edge(&graph, 2, 1, 0.0);
  integer_graph_ins_edge(&graph, 2, 4, 0.0);
  integer_graph_ins_edge(&graph, 2, 5, 0.0);
  integer_graph_ins_edge(&graph, 2, 6, 0.0);
  integer_graph_ins_edge(&graph, 2, 7, 0.0);
  integer_graph_ins_edge(&graph, 3, 4, 0.0);
  integer_graph_ins_edge(&graph, 4, 1, 0.0);
  integer_graph_ins_edge(&graph, 4, 5, 0.0);
  integer_graph_ins_edge(&graph, 6, 4, 0.0);

  print_graph (&graph);
  printf ("\n\n");

  bfs(&graph, 2);

  integer_graph_destroy (&graph);
  return EXIT_SUCCESS;
}
```

Question 4

Mettez en œuvre le parcours en largeur d'un graphe. Le prototype de la fonction sera

```
void bfs(integer_graph_t* graph, int root);
```

Solution:

```
void bfs (integer_graph_t* graph, int root) {
  integer_queue_t queue;
  integer_queue_init (&queue);
  integer_set_t set;
  integer_set_init (&set);

  integer_set_insert (&set, root);
  integer_queue_enqueue (&queue, root);

  while (integer_queue_size (&queue) > 0) {
    int vertex;
    integer_queue_dequeue (&queue, &vertex);

    printf ("Vertex␣%d␣visited\n", vertex);

    generic_list_elmt_t* lelem = generic_list_head (&(graph->adjlists));
    for (; lelem != NULL; lelem = generic_list_next (lelem)) {
      integer_adjlist_t* node = (integer_adjlist_t*) generic_list_data (lelem);
      if (node->vertex == vertex) {


        generic_list_elmt_t* lelem2 = generic_list_head (&(node->adjacent));
        for (; lelem2 != NULL; lelem2 = generic_list_next (lelem2)) {
          integer_adjlist_elmt_t* child =
            (integer_adjlist_elmt_t*) generic_list_data (lelem2);
          if (!integer_set_is_member (&set, child->vertex)) {
            integer_set_insert (&set, child->vertex);
            integer_queue_enqueue (&queue, child->vertex);
          }
        }
      }
    }
  }
}
```

On rappelle l'algorithme du parcours en largeur

```
Function bfs(graph, racine) :
    queue_t q;
    queue_enqueue(q, racine);
    set_t s;
    set_insert(s, racine);
    while queue_size(q) > 0 do
        elem = queue_dequeue(q);
        Afficher(elem);
        for /* Pour tous les voisins de elem */ do
            if set_is_member(s, elem) == false then
                set_insert(s, elem);
                queue_enqueue(q, elem);
            end
        end
    end
```

Explication(s):

- ✿ Le point difficile de cette mise en œuvre de l'algorithme est dans la recherche des voisins qui nécessite deux itérateurs.
  - — Le premier itérateur permet de trouver la liste des voisins d'un nœud.
  - — Le second itérateur, agit sur tous les éléments de l'ensemble, ajoute les nœuds non visités dans la file.
- ✿ Pour marquer les nœuds qui ont déjà été visités, nous utilisons une structure de données ensemble (set). En effet son API a une fonction `insert` et une fonction `is_member` qui sont pratiques pour notre algorithme.

Exercice 2 – Modélisation et parcours

Le matériel de cet exercice se trouve dans le répertoire `exo3`. qL'ennoncé de l'exercice est une traduction de l'exercice donné

`https://www.hackerearth.com/practice/algorithms/graphs/topological-sort/practice-problems/algorithm/oliver-and-the-game-3/`

Oliver et Bob sont les meilleurs amis du monde. Ils ont passé toute leur enfance dans la belle ville de Byteland. Les habitants de Byteland vivent heureux avec le roi.

La ville a une architecture unique avec un total de N maisons. Le manoir du roi est un très grand et magnifique bungalow dont l'adresse est 1. Les autres maisons de Byteland ont une adresse unique (par exemple, A), sont reliées par des routes et il y a toujours un chemin unique entre deux maisons de la ville. Notez que le manoir du roi fait également partie de ces maisons.

Oliver et Bob ont décidé de jouer à cache-cache en prenant la ville entière comme arène. Dans le scénario du jeu, c'est à Oliver de se cacher et à Bob de le trouver.

Oliver peut se cacher dans n'importe quelle maison de la ville, y compris le manoir du roi. Comme Bob est une personne très paresseuse, pour trouver Oliver, soit il se dirige vers le manoir du roi (il s'arrête quand il l'atteint), ou il s'éloigne du manoir par n'importe quel chemin possible jusqu'à la dernière maison sur ce chemin.

Oliver court se cacher dans une maison (par exemple, X) et Bob commence le jeu à partir de sa maison (par exemple, Y). Si Bob atteint la maison X, il trouvera certainement Oliver.

Étant donné Q requêtes, vous devez dire à Bob s'il est possible pour lui de trouver Oliver ou non.

Les requêtes peuvent être des deux types suivants :

— $0\ X\ Y$: Bob se déplace vers le manoir du roi.
— $1\ X\ Y$: Bob s'éloigne du manoir du roi.

ENTRÉE

— La première ligne de l'entrée contient un seul entier N, le nombre total de maisons dans la ville.
— Les $N-1$ lignes suivantes contiennent deux entiers A et B séparés par des espaces, indiquant une route entre les maisons situées aux adresses A et B.
— La ligne suivante contient un seul entier Q représentant le nombre de requêtes.
— Les Q lignes suivantes contiennent trois nombres entiers séparés par un espace, représentant chaque requête comme expliqué ci-dessus.

SORTIE Affichage de "YES" ou "NO", une réponse par ligne, pour chaque requête en fonction de la réponse à la requête.

CONSTRAINTES

— $1 \leqslant N \leqslant 10^5$
— $1 \leqslant A, B \leqslant N$
— $1 \leqslant Q \leqslant 5 \cdot 10^5$
— $1 \leqslant X, Y \leqslant N$

Exemple de fichier d'entrée	Sortie associée
9	YES
1 2	NO
1 3	NO
2 6	NO
2 7	YES
6 9	
7 8	
3 4	
3 5	
5	
0 2 8	
1 2 8	
1 6 5	
0 6 5	
1 9 1	

Explications

- Requête 1 Bob va de 8 vers 1 et rencontre 2 sur son chemin.
- Requête 2 Bob part de 8 en s'éloignant de 1 et ne rencontre jamais 2.
- Requête 3 Bob part de 5 en s'éloignant de 1 et ne rencontre jamais 6.
- Requête 4 Bob va de 5 vers 1 et ne rencontre jamais 6.
- Requête 5 Bob part de 1 en s'éloignant de 1 et rencontre Oliver à 9. Il peut prendre les deux chemins suivants $1 \to 2 \to 6 \to 9$ ou $1 \to 2 \to 7 \to 8$, 9 apparaît dans au moins l'un des deux.

Pensez à dessiner le graphe pour faciliter la compréhension.

Les fichiers exemples sont donnés dans le répertoire `exo3` sous le nom de `digraph-dfs.in` et `digraph-dfs.out`. Pour exécuter le test, utilisez la commande suivante

```
./digraph-dfs.x < digraph-dfs.in
```

et si vous voulez tester la sortie par la même occasion, utilisez la commande

```
./digraph-dfs.x < digraph-dfs.in | diff - digraph-dfs.out
```

Cette dernière commande ne doit rien afficher si la sortie de votre programme et le contenu du fichier `digraph-dfs.out` sont identiques.

Question 1

Comment modéliser la ville et ses parcours depuis ou vers le bungalow du roi ?

Solution:
En général, à l'aide d'un graphe dirigé non valué. Plus précisément, il faut un graphe et son transposé pour avoir deux structures de donnés représentant les chemins possibles entre les maisons dans le sens du parcours choisi.

Question 2

Proposez une solution algorithmique à ce problème. On passera par une fonction auxiliaire nommée `check` dont le prototype sera

```
bool check (integer_graph_t* graph, int start, int place)
```

Cette fonction vérifie s'il est possible d'atteindre la cachette de Oliver représentée par l'argument `place` depuis la maison de Bob représentée par l'argument `start`.

Solution:

Explication(s):

- ✿ On va modéliser le problème par un graphe et résoudre le problème par un parcours de graphe en profondeur d'abord.
- ✿ Plus précisément, nous allons construire un graphe et son transposé afin de pouvoir aller dans un sens et dans l'autre.
- ✿ Remarque : si on y regarde de plus près, le graphe est en fait un arbre.
- ✿ La principale question à laquelle il faut répondre est "s'il est possible pour lui de trouver Oliver ou non" autrement dit s'il existe un chemin du point de départ vers la cachette d'Oliver.
- ✿ Pour chaque requête, on effectue un parcours en profondeur du graphe ou de son transposé et on vérifie si la maison dans laquelle se cache Oliver est dans la liste, si oui il y a un chemin qui permet de trouver Oliver autrement non.
- ✿ Remarque : on aurait pu aussi utiliser un parcours de graghe en largeur mais il faut bien pratiquer les fonctions de la bibliothèque `libin103`.

Première partie de la solution, construction des graphes :

```
integer_graph_t graph_away;
integer_graph_init (&graph_away);
integer_graph_t graph_towards;
integer_graph_init (&graph_towards);

int nbHouses;
fscanf (stdin, "%d", &nbHouses);

for (int i = 1; i <= nbHouses ; i++) {
  integer_graph_ins_vertex(&graph_away, i);
  integer_graph_ins_vertex(&graph_towards, i);
}

for (int i = 1; i <= nbHouses - 1; i++) {
  int start, end;
  fscanf (stdin, "%d␣%d", &start, &end);
  integer_graph_ins_edge(&graph_away, start, end, 0.0);
  integer_graph_ins_edge(&graph_towards, end, start, 0.0);
}
```

Seconde partie de la solution, détectée si la cachette de Oliver est atteignable depuis la maison de Bob et dans le sens de parcours imposé

— Traitement des requêtes

```
  int nbQueries;
  fscanf (stdin, "%d", &nbQueries);

  bool result = false;
  for (int i = 1; i <= nbQueries; i++) {
    int direction;
    int hidePlace;
    int start;
    fscanf (stdin, "%d␣%d␣%d", &direction, &hidePlace, &start);
    if (direction == 1) { // move away
      result = check (&graph_away, start, hidePlace);
    }
    else { // direction == 0 : move towards
      result = check (&graph_towards, start, hidePlace);
    }

    if (result == true) {
      printf ("YES\n");
    }
    else {
      printf ("NO\n");
    }
  }
```

— Fonction auxiliaire pour répondre à la requête

```
bool check (integer_graph_t* graph, int start, int place) {
  integer_list_t ordered_dfs;
  integer_list_init (&ordered_dfs);
  integer_dfs (graph, start, &ordered_dfs);
  integer_list_elmt_t* elem = integer_list_head (&ordered_dfs);
  for (; elem != NULL; elem = integer_list_next (elem)) {
    int n = integer_list_data (elem);
    if (n == place) {
      integer_list_destroy (&ordered_dfs);
      return true;
    }
  }
  integer_list_destroy (&ordered_dfs);
  return false;
}
```

Exercice 1 – Calculer le chemin de sortie d'un labyrinthe

Fonctions utiles pour cet exercice

- `integer_graph_init`
- `integer_graph_destroy`
- `integer_graph_ins_vertex`
- `integer_graph_ins_edge`

Dans l'exercice 2 de la feuille 4 de TP, nous avons développé un algorithme de génération de labyrinthes à l'aide de la structure de données union-find. Dans cet exercice, nous allons mettre en place un parcours de graphe pour trouver le chemin vers la sortie du labyrinthe généré.

Question 1

Quel type de parcours de graphes allons-nous utiliser pour trouver le plus cours chemin ?

Solution:
Un parcours en largeur car nous avons une modélisation fondée sur un graphe non valué.
Explication(s):

- ✿ L'idée est de parcourir de proche en proche les cases libres du labyrinthe afin de trouver la route vers la sortie. On fait une recherche exhaustive des cases libres.

Le matériel pour cet exercice se situe dans le répertoire `exo2`.

Question 2

Pour trouver le chemin dans un labyrinthe avec un parcours en largeur, il faut mémoriser en plus de l'ordre des nœuds visités, son parent afin d'exhiber le chemin de l'entrée vers la sortie.

On écrira une fonction dont le prototype est

```
void vers_la_sortie(integer_graph_t* graph, int entree, int sortie);
```

qui affichera la succession de sommets parcourus pour aller de l'entrée du labyrinthe vers la sortie.

Solution:

```
void vers_la_sortie (integer_graph_t* graph, int entree, int sortie) {
  integer_queue_t path;
  integer_queue_init (&path);
  integer_queue_t queue;
  integer_queue_init (&queue);
  integer_set_t set;
  integer_set_init (&set);

  integer_set_insert (&set, entree);
  integer_queue_enqueue (&queue, entree);

  bool trouve = false;

  while (integer_queue_size (&queue) > 0 && trouve == false) {
    int vertex;
    integer_queue_dequeue (&queue, &vertex);

    generic_list_elmt_t* lelem = generic_list_head (&(graph->adjlists));
    for (; lelem != NULL; lelem = generic_list_next (lelem)) {
      integer_adjlist_t* node = (integer_adjlist_t*) generic_list_data (lelem);
      if (node->vertex == vertex) {
        if (node->vertex == sortie) {
          trouve = true;
          integer_queue_enqueue (&path, node->vertex);
        }
        else {
          generic_list_elmt_t* lelem2 = generic_list_head (&(node->adjacent));
          for (; lelem2 != NULL; lelem2 = generic_list_next (lelem2)) {
            integer_adjlist_elmt_t* child =
              (integer_adjlist_elmt_t*) generic_list_data (lelem2);

            if (!integer_set_is_member (&set, child->vertex)) {
              integer_set_insert (&set, child->vertex);
              integer_queue_enqueue (&queue, child->vertex);
              integer_queue_enqueue (&path, node->vertex);
            }
          }
        }
      }
    }
  }
  if (trouve == false) {
    printf ("Il n'y a pas de chemin entre %d et %d\n", entree, sortie);
  }
  else {
    integer_list_elmt_t* elem = integer_list_head (&path);
    int pred = integer_list_data(elem);
    printf ("%d -> ", pred);
    elem = integer_list_next (elem);
    for (; elem != NULL; elem = integer_list_next (elem)) {
      int d = integer_list_data(elem);
      if (d != pred) {
        pred = d;
        printf ("%d -> ", d);
      }
    }
    printf ("\n");
  }
}
```

Explication(s):

- ✿ L'idée sous-jacente est d'utiliser une nouvelle structure de données pour stocker la liste des parents. Plus spécifiquement, nous utilisons une file (`path`) pour enregistrer les parents des nœuds visités (cf ligne 36 de la correction). Il y aura nécessairement répétition de sommets dans la file qui pourront être filtrés plus tard lors de l'affichage.
- ✿ Par rapport à la version du parcours en largeur simple, nous ajoutons une condition d'arrêt précoce quand le sommet visité est le sommet « arrivé » que nous cherchons (ligne 28 de la correction). Dans ce cas il ne faut pas oublier d'ajouter le sommet final dans la file (ligne 30).
- ✿ Nous avons ajouté une variable drapeau (`trouve`) qui permet d'arrêter prématurément la boucle `while` (cf ligne 14 et ligne 29) quand le sommet cherché (c.-à-d., la sortie du labyrinthe) a été trouvé.

Question 3

La fonction `generate_maze` est une modification de l'algorithme de génération du labyrinthe demandé dans la feuille 4. La principale différence est qu'elle renvoie (par effet de bord) la liste des cases adjacentes dans le labyrinthe. En d'autres termes, au lieu de renvoyer la liste des murs du labyrinthe, la fonction renvoie la liste des cases qui forment les couloirs du labyrinthe.

1. Expliquer comment sont calculées les cases des couloirs.

2. Utiliser cette fonction `generate_maze` pour construire le graphe associé aux couloirs du labyrinthe généré et appliquer la fonction `vers_la_sortie` pour trouver le chemin entre la case de départ 0 et la case d'arrivée numéro dimension$^2-1$.

Solution:

1. Les cases qui forment les couloirs sont données par les murs qui ont été supprimés lors de la génération du labyrinthe.
2. Dans la fonction `generate_maze` une nouvelle variable `freePaths` a été utilisée pour stocker les murs qui ont été ignorés lors de la génération du labyrinthe. Ces mûrs donnent les cases qui forment les couloirs.

```
*paths = malloc ((lenFreePaths) * sizeof(edge_t));
*size = lenFreePaths;
for (k = 0; k < lenFreePaths; k++) {
  (*paths)[k] = freePaths[k];
}
```

2 Deux étapes :
   — Créer autant de sommets que de cases dans la grille utilisée pour le labyrinthe
   — Ajouter les arêtes du graphes à partir du résultat donné par la fonction `generate_maze`. Cette fonction retourne les murs qui ont été supprimés (c.-à-d., pas considérer comme mur final pour le labyrinthe), ils nous donnent les numéros des cases qui sont accessibles entres elles.

1. **Attention** nous manipulons un graphe non orienté dans cet exercice. Il faut faire l'insertion d'une arête (a,b) et également une insertion de l'arête (b,a).

```
int main (int argc, char** argv) {

  int dimension = 4;
  int nbCases = dimension * dimension;
  edge_t* path = NULL;
  int len;

  generate_maze (dimension, &path, &len);

  integer_graph_t graph;
  integer_graph_init (&graph);

  for (int i = 0; i < nbCases; i++) {
    integer_graph_ins_vertex(&graph, i);
  }

  for (int k = 0; k < len; k++) {
    printf ("(%d,␣%d)␣-␣", path[k].cell1, path[k].cell2);
    integer_graph_ins_edge(&graph, path[k].cell1, path[k].cell2, 0.0);
    // integer_graph_ins_edge(&graph, path[k].cell2, path[k].cell1, 0.0);
  }
  printf ("\n\n");
  print_graph (&graph);
  printf ("\n\n");

  printf ("Le␣chemin␣de␣longueur␣minimal␣allant␣de␣%d␣vers␣%d␣est␣:␣", 0, nbCases-1);
  vers_la_sortie (&graph, 0, nbCases-1);

  integer_graph_destroy (&graph);

  return EXIT_SUCCESS;
}
```

APPROFONDISSEMENT

Exercice 1 – Coloration de graphe

Le problème de colorier un graphe est de positionner des couleurs à chaque sommet du graphes tels que les voisins ont une couleur différente.

Le matériel pour cet exercice se situe dans le répertoire `exo4`.

Question 1

Supposons un graphe $G=(V,E)$ tel que $\mid V \mid = n$. On suppose que les nœuds sont représentés par des entiers positifs, c'est-à-dire, $V=\{1,2,\ldots,n\}$. On impose donc un ordre sur le nœuds du graphe. Les couleurs sont également représentées par des entiers positifs. Le nombre minimum de couleurs nécessaire pour colorier un graphe est nommé

nombre chromatique du graphe. C'est un problème NP-difficile pour lequel plusieurs algorithmes ont été proposés dans la littérature.

Notons $c(x)$ la couleur associée au nœud x. Imaginez un algorithme glouton pour colorier un graphe.

Solution:

$k \leftarrow 0$;
for $i \leftarrow 1$ **to** n **do**
  Soit $c(i)$ le plus petit entier tel que $c(i) \notin \{c(j) : j \in \text{adj}(i)\}$;
  if $c(i) > k$ **then**
    $k \leftarrow c(i)$;
  end
end
return k;

Question 2

Développez une fonction nommée `greedyColoring` qui permet de colorier un graphe et dont le prototype est :

```
int greedyColoring (integer_graph_t* graph, int** colors);
```

On supposera que le premier sommet du graphe est 0, le tableau `colors` sera alloué dans la fonction `greedyColoring`.

Solution:

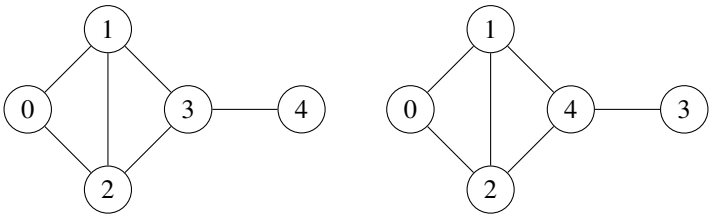
```
int greedyColoring (integer_graph_t* graph, int** colors) {
  *colors = malloc (sizeof(int) * integer_graph_vcount(graph));
  if (*colors == NULL) {
    return -1;
  }
  for (int i = 0; i < integer_graph_vcount(graph); i++) {
    (*colors)[i] = -1;
  }

  bool* available = malloc (sizeof(bool) * integer_graph_vcount(graph));
  if (available == NULL) {
    return -1;
  }

  (*colors)[0] = 0;

  for (int vertex = 1; vertex < integer_graph_vcount (graph); vertex++) {
    for (int i = 0; i < integer_graph_vcount(graph); i++) {
      available[i] = false;
    }

    integer_list_t* neighbors = integer_graph_adjlist (graph, vertex);
    integer_list_elmt_t* elem = integer_list_head (neighbors);
    for (; elem != NULL; elem = integer_list_next (elem)) {
      int n = integer_list_data (elem);
      if ((*colors)[n] != -1) {
        available[(*colors)[n]] = true;
      }
    }

    int c;
    for (c = 0; c < integer_graph_vcount(graph); c++) {
      if (available[c] == false) {
        break;
      }
    }

    (*colors)[vertex] = c;
  }

  free(available);
  return 0;
}
```

Explication(s):

- Principe de la mise en oeuvre
  - on va parcourir tous les sommest du graphes
  - pour chaque voisin d'un sommet on regarde s'il est déjà colorié ou pas et on conserve cette information dans un tableau de couleurs
  - une fois les voisins parcourus, on cherche la premiere couleur libre dans le tableau, c'est cette couleur que l'on donne au sommet
- Lignes 2 à 12 on alloue le tableau `colors` et le tableau `available` qui contient la liste des couleurs possibles. On maximum on a une couleur par sommets.
- Ligne 14, on affecte la première couleur au premier sommet du graphe supposé être 0.
- Lignes 16 à 39, la boucle principale de l'algorithme qui parcours chaque sommet du graphe
- Lignes 18 à 19, on initialize le tableau des couleurs à false indiquant qu'aucune couleur n'est prise.
- Lignes 22 à 29, pour chaque voisin d'un sommet on regarde s'il est déjà colorié ou non et on modifie le tableau `available`
- Lignes 31 à 38, on parcourt le tableau `available` pour trouver le première couleur libre et on l'affecte au sommet courant.

Question 3

Testez votre fonction sur les graphes suivants :



Que constatez-vous ?

Solution:
Explication(s):

- ✿ En utilisation la notation $x \to c$, le sommet x a la couleur c
- ✿ Pour le premier graphe, on a : $0 \to 0$, $1 \to 1$, $2 \to 2$, $3 \to 0$, $4 \to 1$
- ✿ Pour le second graphe, on a : $0 \to 0$, $1 \to 1$, $2 \to 2$, $3 \to 0$, $4 \to 3$
- ✿ La coloration des graphes par la méthode gloutonne dépend de la construction du graphe.

Question 4 – Application

Nous voulons résoudre le problème donné

`https://open.kattis.com/problems/vivoparc`

Les fichiers exemples sont donnés dans le répertoire `exo4` sous le nom de `vivoparc.in` et `vivoparc.out`. Pour exécuter le test, utilisez la commande suivante

```
./vivoparc.x < vivoparc.in
```

et si vous voulez tester la sortie par la même occasion, utilisez la commande

```
./vivoparc.x < vivoparc.in | diff - vivoparc.out
```

Cette dernière commande ne doit rien afficher si la sortie de votre programme et le contenu du fichier `vivoparc.out` sont identiques.

Solution:

```
int main (int argc, char** argv) {
  integer_graph_t graph;
  integer_graph_init (&graph);

  int nbEnclosures;
  fscanf (stdin, "%d", &nbEnclosures);

  for (int i = 0; i < nbEnclosures ; i++) {
    integer_graph_ins_vertex(&graph, i);
  }

  int startEnclosure;
  int endEnclosure;
  while (fscanf(stdin, "%d-%d", &startEnclosure, &endEnclosure) != EOF) {
    integer_graph_ins_edge(&graph, startEnclosure-1, endEnclosure-1, 0.0);
    integer_graph_ins_edge(&graph, endEnclosure-1, startEnclosure-1, 0.0);
  }

  int* colors;
  greedyColoring(&graph, &colors);

  for (int i = 0; i < integer_graph_vcount (&graph); i++) {
    printf ("%d␣%d\n", i+1, colors[i]+1);
  }

  free(colors);
  integer_graph_destroy (&graph);
  return EXIT_SUCCESS;
}
```

Explication(s):

- ✿ Il suffit d'appliquer la fonction `greedyColoring` sur une modélisation du problème à l'aide d'un graphe.
- ✿ La construction des graphes dans la bibliothèque `libin103` évite déjà l'insertion mutiple d'arc entre le même couple de sommets.
- ✿ Il faut penser à ajouter les arcs deux fois, une fois dans un sens et une fois dans l'autre pour avoir un graphe non orienté.
- ✿ La seule astuce est que la notation des enclos débute à 1 et les sommets de nos graphes à 0 d'où le décalage de 1 dans un sens à la lecture et et dans l'autre sens à l'affichage du résultat. Idem pour les couleurs.