Devoir Non Surveillé 1

À rendre le 4 novembre 2024

INFORMATIQUE MPI/MPI*

Vue d'ensemble du sujet

Ce sujet est composé de 3 parties indépendantes, utilisant les langages de programmation C et OCaml.

Les différentes parties sont indépendantes et peuvent être traités dans n'importe quel ordre. Il n'est pas nécessaire d'avoir répondu aux questions d'une partie pour répondre aux questions d'une autre partie.

- La partie I se concentre sur l'étude des couplages de cardinal maximum, leur lien avec les couvertures par sommet, puis propose des jeux à 2 joueurs sur ce thème.
- La partie II s'intéresse aux tests de primalité probabilistes, et à leur implémentation en C.
- La partie III s'intéresse à l'analyse et l'implémentation d'un algorithme de Tarjan pour trouver des plus petits ancêtres communs dans un arbre.

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat. En langage C, il est inutile de rappeler que les entêtes `<assert.h>`, `<stdbool.h>`, etc. doivent être inclus.

Quand l'énoncé demande de coder une fonction, sauf indication explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle. Lorsqu'une réponse en pseudo-code est permise, seule la logique de programmation est évaluée, même dans le cas où un code en C a été fourni en guise de réponse.

Partie I. Couplages dans les graphes bipartis

On s'intéresse au problème de la *couverture minimale par sommets* (ou problème du transversal minimum) : étant donné un graphe $G = (S, A)$, déterminer le nombre de sommets minimal d'un ensemble $C \subset S$ tel que $\forall \{x, y\} \in A, x \in C \vee y \in C$. Un tel ensemble est appelé une couverture par sommets du graphe G.

☐ 1 – Démontrer que le nombre de sommets d'une couverture minimale par sommets est toujours supérieur ou égal au nombre d'arêtes d'un couplage de cardinal maximum.

Réponse 1. Soit C une couverture par sommets et M un couplage. Par définition de C, on a $\forall a \in M. \exists x \in a. x \in C$. Donc si on note $|M| = m$, $\exists (x_i)_{i=1\dots m}$, $x_i \in C$, et par définition de M, les x_i sont deux à deux distincts, donc $|\{x_i, i = 1 \dots m\}| = m$, et $|C| \geq m$. En particulier, c'est vrai pour une couverture minimale par sommets et un couplage de cardinal maximum.

Dans la suite de cette partie, on s'intéresse à un graphe biparti non orienté $G = (S, A)$, avec $S = X \cup Y$, $X \cap Y = \emptyset$ et $A \subset \{ \{x, y\} | \ x \in X, y \in Y\}$.

Soit M un couplage de cardinal maximum pour G. On note L l'ensemble des sommets accessibles par un chemin alternant [1] depuis un sommet libre de X.

☐ 2 – Montrer que $C = (X \setminus L) \cup (Y \cap L)$ constitue une couverture par sommets de G.

Réponse 2. Analysons d'abord un peu les chemins alternants depuis un sommet libre de X. Un tel chemin est de la forme :

$$x_0 - y_0 - x_1 - \dots$$

On a nécessairement $\{x_0, y_0\} \notin M$ car x_0 est libre. Par récurrence immédiate, on a pour ce chemin : $\forall i, \{x_i, y_i\} \notin M$ et $\{y_i, x_{i+1}\} \in M$.

Revenons à la question sur C. Soit $a = \{x, y\} \in A$ une arête quelconque, avec $x \in X$ et $y \in Y$. On va montrer que $x \in L \Rightarrow y \in L$. Procédons par disjonction de cas :

$a \in M$ **:** Tout chemin alternant depuis un sommet libre de X terminant en x se termine par une arête du couplage, donc a. Donc si $x \in L$, alors nécessairement $y \in L$, en suivant le même chemin sans l'arête a.

$a \notin M$ **:** Tout chemin alternant depuis un sommet libre de X terminant en x se termine par une arête du couplage, donc on peut le prolonger avec l'arête a pour obtenir un nouveau chemin alternant terminant en y, et $y \in L$.

Dans tous les cas, on a soit $x \notin L$ et donc $x \in C$, soit $y \in L$ et donc $y \in C$. C est donc bien une couverture par sommets.

☐ 3 – Justifier que L ne contient aucun sommet libre de Y.

Réponse 3. Si L contient un sommet libre de Y, on a un chemin alternant joignant un sommet libre de X et un sommet libre de Y, c'est-à-dire un chemin augmentant, ce qui contredit la maximalité de M.

☐ 4 – Montrer qu'il n'existe pas d'arête de M reliant un sommet de $X \setminus L$ et un sommet de $Y \cap L$.

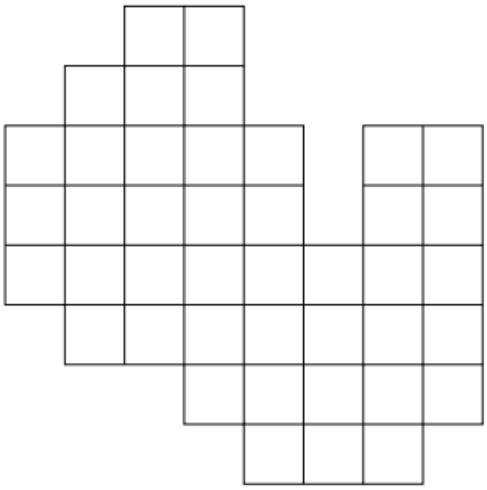
Réponse 4. Par l'absurde, soit $a = \{x, y\} \in M$, avec $x \in X \setminus L$ et $y \in Y \cap L$. Alors il existe un chemin alternant joignant un sommet libre de X et y. Comme vu précédemment, ce chemin se termine nécessairement par une arête hors de M. Donc on peut étendre ce chemin avec a en un chemin alternant joignant un sommet libre de X et x, et $x \in L$, ce qui est une contradiction.

☐ 5 – Démontrer le théorème de König (1931) :
Pour tout graphe biparti, le cardinal maximum d'un couplage est égal au cardinal minimum d'une couverture par sommets.

Réponse 5. C ne contient aucun sommet libre, donc chaque arête de M possède au moins un sommet dans C. De plus, aucune arête de M ne possède ses deux sommets dans C (question 4). Donc chaque arête de M possède exactement un sommet dans C, qui sont tous différents, et $|C| = |M|$.

☐ 6 – Peut-on paver la figure suivante à l'aide de dominos ? Donner un pavage ou une preuve concise qu'aucun tel pavage ne peut exister.

1. On appelle *chemin* alternant un chemin qui alterne les arêtes dans M et les arêtes dans $A \setminus M$



Réponse 6. Si on colorie les cases de la figure en blanc et noir comme un damier, on peut remarquer qu'un domino couvre toujours une case noire et et une case blanche. On a donc bien un graphe biparti. Une couverture par des dominos correspond à un couplage dans ce graphe car chaque case ne peut être recouverte que par un seul domino. Il y a 42 cases. On peut trouver une couverture du graphe avec 20 sommets, donc on ne peut pas recouvrir toute la figure à l'aide de dominos, ce qui constituerait un couplage parfait de 21 dominos.

☐ 7 – Démontrer le théorème de Hall (1935) :
G possède un couplage de cardinal $|X|$ si et seulement si pour tout $P \subset X$, on a $|V(P)| \geq |P|$
où $V(P) = \{y \in Y \mid \exists x \in P, \{x, y\} \in A\}$ est le voisinage de P.

Réponse 7. Soit M un couplage de cardinal $|X|$. Considérons $\{y \in Y \mid \exists x \in P, \{x, y\} \in M\}$.
Cet ensemble est inclus dans $V(P)$, et de cardinal $|P|$ par définition d'un couplage.
Soit C une converture par sommets minimale, dont le nombre de sommets dans X est maximal. S'il n'existe pas de couplage de cardinal $|X|$, alors $|C| < |X|$. Soit $x \in X \setminus C$, si x est isolé, alors $P = \{x\}$ vérifie $|V(P)| < |P|$. Sinon, $C \cap Y \neq \emptyset$, et $P = \{x \mid \exists y \in C, \{x, y\} \in A\}$ vérifie $|V(P)| < |P|$ par minimalité/maximalité de C.

☐ 8 – On considère le jeu à 2 joueurs suivant sur un graphe (pas nécessairement biparti) $G = (S, A)$. Les joueurs 1 et 2 choisissent chacun leur tour une arête pour former un chemin élémentaire avec les arêtes déjà sélectionnées (on rappelle qu'un chemin élémentaire ne peut pas passer plusieurs fois par le même sommet). Le premier joueur qui ne peut pas choisir d'arête a perdu. Montrer que si G possède un couplage parfait, le joueur 1 possède une stratégie gagnante.

Réponse 8. La stratégie du joueur 1 consiste à choisir toujours une arête du couplage parfait.
Pour prolonger un chemin, le joueur 2 ne peut pas choisir une arête du couplage parfait, puisque les deux extrémités du chemin terminent par une arête du couplage. Le joueur 1 en revanche, peut toujours choisir de prolonger le chemin en ajoutant à la suite de l'arête choisie par le joueur 2 une arête du couplage. Puisque le joueur 1 peut toujours jouer, et que la partie ne peut pas être infinie, il gagne nécessairement.

☐ 9 – On considère le jeu à 2 joueurs suivant sur un graphe biparti. On commence avec un couplage vide $M := \emptyset$. Chacun leur tour, les joueurs choisissent un chemin augmentant I pour le couplage M, et on met à jour $M := M \oplus I$, où $\oplus$ est la différence symétrique. Si un joueur ne peut plus jouer, il perd et l'autre joueur gagne. Déterminer si un joueur possède une stratégie gagnante, et si oui lequel.

Réponse 9. On peut remarquer que pour tout chemin augmentant I pour un couplage M, $|M \oplus I| = |M| + 1$.
Donc quels que soient les choix des joueurs, la partie se terminera au bout de n tours, où n est le cardinal d'un couplage de cardinal maximal. Si n est impair, le joueur 1 gagne, si n est pair, le joueur 2 gagne.

Partie II. Test de primalité (C)

Dans cette partie, on utilisera le langage C en supposant qu'il n'y a pas de dépassement de capacité sur les entiers, et que la fonction `rand()` renvoie un nombre aléatoire entre 0 et M, où M est suffisamment grand pour que `rand() % k` renvoie un nombre uniformément aléatoire entre 0 et $k-1$.

Test de primalité de Fermat

Si un nombre n est premier, alors tout nombre a premier avec n vérifie $a^{n-1} = 1 \mod n$. Le test de primalité de Fermat est un algorithme probabiliste basé sur cette observation.

☐ 10 – Écrire une fonction `int mod_exp(int a, int b, int n)` qui calcule $a^b \mod n$ en complexité en temps $O(\log(b))$, en supposant que l'addtion et la multiplication d'entiers s'effectue en temps constant.

Réponse 10.

C
```c
int mod_exp(int a, int b){
  int acc = 1;
  while (b > 0){
    if (b % 2 == 1) {
      acc = acc * a % n;
    }
    a = a * a % n;
    b = b / 2;
  }
  return acc;
}
```

☐ 11 – Écrire une fonction `bool fermat_witness(int n, int a)` qui renvoie `true` si a est un témoin de primalité de Fermat de n (i.e., $a^{n-1} = 1 \mod n$) et `false` sinon.

Réponse 11.

C
```c
bool fermat_witness(int n, int a){
  return mod_exp(a, n-1, n) == 1;
}
```

☐ 12 – Écrire une fonction `bool fermat_primality_test(int n, int k)` qui tire aléatoirement k nombres entre 2 et $n-1$ inclus et qui renvoie `true` si tous ces nombres sont des témoins de primalité de Fermat de n, et `false` si au moins l'un d'entre eux ne l'est pas.

Réponse 12.

C
```c
bool fermat_primality_test(int n, int k){
  for (int i = 0; i < k; i = i + 1){
    int a = 2 + (rand() % (n-2));
    if (!fermat_witness(n, a)) {
      return false;
    }
  }
  return true;
}
```

Cet algorithme se révèle très efficace, car la moitié des nombres a entre 2 et $n-1$ sont des témoins de non-primalité si n n'est pas premier, sauf pour une famille de nombres particuliers : les nombres de Carmichael. Le test suivant permet d'éviter ces faux nombres premiers.

Test de primalité de Miller-Rabin

Soit p un nombre premier impair. Notons que 1 et -1 sont les seules racines carrées de 1 modulo p. Soit s non nul et d impair deux entiers vérifiant $p-1=2^s d$. Alors pour tout entier a qui n'est pas un multiple de p, on a : $a^{p-1}=1 \mod p$. Donc

$$a^d = 1 \mod p \quad \vee \quad \exists r \in [\![0, s-1]\!], \quad a^{2^r d} = -1 \mod p.$$

Donc (par contraposée) pour tout entiers s non nul, d impair, $n=2^s d+1$ et $1<a<n$,

$$\left(a^d \neq 1 \mod n \wedge \forall r \in [\![0, s-1]\!], a^{2^r d} \neq -1 \mod n\right) \to n \text{ est composite.}$$

Dans ce cas, on appelle a un témoin de non-primalité de Miller pour n.

☐ 13 – Écrire une fonction `void decompose(int n, int *s, int *d)` qui à partir d'un nombre n donné, calcule les valeurs de s et de d et change les valeurs pointées par `s` et `d`.

Réponse 13.

C

```c
void decompose(int n, int *s, int *d){
  *s = 0;
  *d = n - 1;
  while (*d % 2 == 0){
    *s = *s + 1;
    *d = *d / 2;
  }
}
```

☐ 14 – Écrire une fonction `bool miller_witness(int n, int a)` qui étant donné n et a renvoie `true` si a est un témoin de non-primalité de n, et `false` sinon.

Réponse 14.

C

```c
bool miller_witness(int n, int a){
  int s, d;
  decompose(n, &s, &d);
  /* Remarque : on aurai pu prendre s et d en argument pour éviter
     de les recaculter à chaque test */
  int acc = mod_exp(a, d, n);
  if (acc == 1) { return false; }
  for (int i = 0; i < s; i = i + 1){
    if (acc == -1) { return false; }
    acc = (acc * acc) % n;
  }
  return true;
}
```

Si n est un nombre impair composé, au moins 3/4 des entiers entre 2 et $n-1$ inclus sont des témoins de non-primalité de Miller pour n. On peut donc en déduire un algorithme de type Monte-Carlo dont la probabilité de correction est supérieure à 0.95.

☐ 15 – Écrire une fonction `bool miller_rabin95(int n)` qui renvoie `true` si n est premier, et qui renvoie `false` avec une probabilité supérieure à 0.95 si n est composé.

Réponse 15. Si n est composé, un seul test de `miller_witness` avec un a aléatoire donnera un mauvais résultat avec une probabilité au plus $\frac{1}{4}$, donc avec 3 tests indépendants, notre algorithme se trompera avec une probabilité $\frac{1}{64} < 0.05$.

C

```c
bool miller_rabin95(int n){
  assert(n > 0);
  if (n <= 1) return false; // 0 et 1 ne sont pas premiers
  if (n <= 3) return true; // 2 et 3 sont premiers
  if (n % 2 == 0) return false; // les nombres pairs > 2 ne sont pas premiers
  for (int i = 0; i < 3; i = i + 1){
    int a = 2 + (rand() % (n-2));
    if (miller_witness(n, a)) {
      return false;
    }
  }
  return true;
}
```

Ce test de primalité à été proposée par Michael Rabin comme variante de l'algorithme proposé par Gary L. Miller dont la preuve de correction repose sur l'hypothèse de Riemann généralisée. Il s'agit dans la version originale de tester les entiers compris entre 2 et $O(\log(n)^2)$.

Partie III. Tarjan's off-line lowest common ancestors algorithm

Cet algorithme, du à Robert Tarjan[2], sert à calculer dans un arbre $\mathcal{T}$, pour une ensemble de paires de nœuds $P = (\{u, v\})$ les plus petit ancêtre communs de u et v. Il est donné par le pseudo-code suivant :

```
function TarjanOLCA(u) is
    MakeSet(u)
    u.ancestor := u
    for each v in u.children do
        TarjanOLCA(v)
        Union(u, v)
        Find(u).ancestor := u
    u.color := black
    for each v such that {u, v} in P do
        if v.color == black then
            print "Tarjan's Lowest Common Ancestor of " + u +
                  " and " + v + " is " + Find(v).ancestor + "."
```

On cherche à le comprendre, et à l'implémenter en OCaml, avec le type suivant pour les arbres (on se restreint à des arbres binaires localement complets non vides) :

```
type 'a tree = Leaf of 'a | Node of ('a tree * 'a * 'a tree)
```

☐ 16 – Proposer un type adapté pour représenter l'ensemble P et justifier votre choix.

Réponse 16. On souhaite efficacement parcourir pour un u donné $\{v \mid \{u, v\} \in P\}$, donc on peut utiliser des listes d'adjacences : `int list array`, ou `('a, 'a list) Hashtbl.t` si les éléments ne sont pas des entiers entre 0 et $n-1$.

2. Un autre algorithme porte le nom de Tarjan, et sert à calculer les composantes fortement connexes dans un graphe orienté.

☐ 17 – Quel type de parcours d'arbre peut-on reconnaître dans cet algorithme ? Justifier.

Réponse 17. Il s'agit d'un parcours en profondeur car le parcours des fils de u termine avant le parcours de u.

☐ 18 – Pour représenter les champs `ancestor` et `color` présentés dans le pseudo-code, nous souhaiterions utiliser des tableaux. Or, nous avons besoin pour cela que les nœuds possèdent une étiquette unique entre 0 et $n-1$, où n est la taille de l'arbre.
Écrire une fonction `label (tree : 'a tree) : (int * 'a) tree` qui prend en entrée un arbre $\mathcal{T}$ et qui renvoie un arbre $\mathcal{T}'$ qui contient les mêmes nœuds que $\mathcal{T}$ mais dont l'étiquette contient en plus un unique identifiant entier entre 0 et $n-1$, où $n-1$ est la taille de l'arbre.

Réponse 18. On s'aide d'une fonction auxiliaire qui prend en paramètre supplémentaire le prochain numéro à utiliser dans la numérotation et renvoie le premier numéro qu'il n'a pas utilisé.

```
let label tree =
  let rec label_aux tree next = match tree with
    | Leaf x -> Leaf (next, x), next + 1
    | Node (g, r, d) ->
      let g, root = label_aux g next in
      let d, next = label_aux d (root + 1) in
      Node (g, (root, r), d), next
  in
  label_aux tree 0
```

☐ 19 – Étudier la complexité de l'algorithme de Tarjan.

Réponse 19. La fonction `TarjanOLCA` est appelée sur tous les nœuds de l'arbre.
Dans un premier temps, on ignore le parcours de P.
En ignorant les appels récursifs, le coût pour chaque nœud est dominé par les opérations `Union` et `Find`, en $O(\alpha(n))$, où n est le nombre de nœuds et $\alpha(n)$ est la complexité amortie d'une opération `Union` ou `Find`. Avec une bonne implémentation, on a $\alpha(n)$ qui est la fonction inverse d'Ackermann. Pour ces opérations, on a un coût total en $O(n\alpha(n))$.
Ensuite, on parcours les voisins de u dans P. Avec une implémentation par listes d'adjacence, on parcours ainsi chaque paire $\{u, v\}$ 2 fois, sans tests supplémentaires. Pour la moitié de celles-ci, on effectue une opération `Find` en $O(n)$. On en déduit donc une complexité $O(|P|\alpha(n))$ pour l'ensemble des parcours dans P.

La complexité totale est donc de $O((n+|P|)\alpha(n))$.

Étudions sa correction

☐ 20 – Démontrer que chaque paire $\{u, v\}$ apparaît une et une seule fois dans l'affichage (`print`) proposé par l'algorithme décrit en pseudo-code.

Réponse 20. La fonction `TarjanOLCA` est appelée une et une seule fois par nœud de l'arbre.
Soit $\{u, v\} \in P$, tel que u devient noir avant v. Alors la paire $\{u, v\}$ n'est affichée que par l'appel de `TarjanOLCA` sur v.

☐ 21 – Démontrer que tant que le plus petit ancêtre commun à u et v n'est pas noir, il est l'ancêtre (`ancestor`) du représentant (`Find`) du premier des deux nœuds rencontrés parmi u et v.

Réponse 21. Soit $\{u, v\} \in P$ tel que u devient noir avant v, et soit r le plus petit ancêtre commun à u et v. u est descendant d'un fils u' de r dont l'exploration est terminée lorsque l'on explore v : en effet, sinon u' serait un ancêtre commun à u et v plus petit que r. Par récurrence immédiate, on peut montrer que tous les nœuds sur la branche qui lie r à u ont été unis, et à la fin de l'exploration de u', l'ancêtre de leur représentant est r. Au moment on l'on rencontre v, c'est donc le cas, car il ne sera modifié qu'à la fin de l'appel sur le nœud r, qui termine nécessairement après celui sur v car v est un descendant de r.

☐ 22 – Proposer une implémentation de cet algorithme de Tarjan.

Réponse 22.

```
let lbl = function
  | Leaf x | Node (_, x, _) -> x

let tarjan_lca tree queries =
  let ancestor = Hashtbl.create 16 in
  let visited = Hashtbl.create 16 in
  let results = Hashtbl.create 16 in
  let uf = UnionFind.create () in

  let rec dfs = function
    | Leaf x ->
        UnionFind.make_set uf x;
        Hashtbl.add ancestor x x;
        Hashtbl.replace visited x true
    | Node (left, x, right) ->
        UnionFind.make_set uf x;
        Hashtbl.add ancestor x x;
        dfs left;
        UnionFind.union uf x (lbl left);
        dfs right;
        UnionFind.union uf x (lbl right);
        Hashtbl.replace visited x true;
        List.iter
          (fun y ->
            let lca = Hashtbl.find ancestor (UnionFind.find uf y) in
            print_endline ("Tarjan's Lowest Common Ancestor of " ^ x ^
                           " and " ^ y ^ " is " ^ lca ^ "."))
          (Hashtbl.find queries x)
  in
  dfs tree
```

Fin de l'épreuve