Calcul de Triangles - Correction

Samy Jaziri

*Le corrigé est proposé en **C**. Mais cela demande beaucoup plus de temps que le OCaml et n'est pas un choix judicieux pour un concours.*

1 Mise en jambe

Les triangles du graphe Γ sont :

$$(1,4,5),(1,2,5),(2,4,5),(1,2,4),(2,3,5)$$

On définit les types utilisés dans ce sujet.

```c
typedef unsigned int sommet;
typedef unsigned int uint;

typedef struct triangle {
    sommet s0;
    sommet s1;
    sommet s2;
} triangle;
```

```c
typedef struct ml_sommets {
    struct ml_sommets* suivant;
    sommet sommet;
} ml_sommets;
typedef ml_sommets* liste_sommets;

typedef struct ml_triangles {
    struct ml_triangles* suivant;
    triangle* triangle;
} ml_triangles;
typedef ml_triangles* liste_triangles;
```

```c
typedef struct liste_adjacence {
    liste_sommets* voisins;
    uint nb_sommets;
} liste_adjacence;

typedef struct matrice_adjacence {
    bool** matrice;
    uint nb_sommets;
} matrice_adjacence;
```

On définit ensuite les fonctions utiles pour chaque structures de données.

Triangles :

```c
triangle* creer_triangle(sommet s0, sommet s1, sommet s2) {
    triangle* t = (triangle*) malloc(sizeof(triangle));
    t->s0 = s0;
    t->s1 = s1;
    t->s2 = s2;
    return t;
}
```

```c
void detruire_triangle(triangle* t) {
    free(t);
}
```

Liste de sommets :

```c
liste_sommets creer_liste_sommets() {
    liste_sommets l = NULL;
    return l;
}
```

```c
void detruire_liste_sommets(liste_sommets l) {
    if(l) {
        detruire_liste_sommets(l->suivant);
        free(l);
    }
}
```

```
uint longueur_liste_sommets(liste_sommets l) {
    uint longueur = 0;
    for (ml_sommets* m = l; m; m = m->suivant)
        ++longueur;
    return longueur;
}
```

```
void ajouter_sommet(liste_sommets* l, sommet s) {
    ml_sommets* m = (ml_sommets*) malloc(sizeof(ml_sommets));
    m->sommet = s;
    m->suivant = *l;
    *l = m;
}
```

```
void inverser_liste_sommets(liste_sommets* l) {
    if (!l || !(*l) || !(*l)->suivant)
        return;
    ml_sommets* p = *l;
    *l = (*l)->suivant;
    p->suivant = NULL;
    ml_sommets* tmp = (*l)->suivant;
    (*l)->suivant = p;
    while(tmp) {
        p = *l;
        *l = tmp;
        tmp = (*l)->suivant;
        (*l)->suivant = p;
    }
}
```

Liste de triangles :

```
liste_triangles creer_liste_triangles() {
    liste_triangles l = NULL;
    return l;
}
```

```
void detruire_liste_triangles(liste_triangles l) {
    if(l) {
        detruire_liste_triangles(l->suivant);
        detruire_triangle(l->triangle);
        free(l);
    }
}
```

```
void ajouter_triangle(liste_triangles* l, triangle* t) {
    ml_triangles* m = (ml_triangles*) malloc(sizeof(ml_triangles));
    m->triangle = t;
    m->suivant = *l;
    *l = m;
}
```

Listes d'adjacence :

```
liste_adjacence* creer_liste_adjacence(uint nb_sommets) {
    liste_adjacence* la = (liste_adjacence*) malloc(sizeof(liste_adjacence));
    la->nb_sommets = nb_sommets;
    la->voisins = (liste_sommets*) malloc(sizeof(liste_sommets) * nb_sommets);
    for (sommet i = 0; i < nb_sommets; ++i)
        la->voisins[i] = creer_liste_sommets();
    return la;
}
```

```
void detruire_liste_adjacence(liste_adjacence* l) {
    if(l) {
        for (sommet i = 0; i < l->nb_sommets; ++i)
            detruire_liste_sommets(l->voisins[i]);
        free(l);
    }
}
```

```
void ajouter_voisins_liste(liste_adjacence* l, sommet s0, sommet s1) {
    ajouter_sommet(&l->voisins[s0], s1);
    ajouter_sommet(&l->voisins[s1], s0);
}
```

Matrice d'adjacence :

```
matrice_adjacence* creer_matrice_adjacence(uint nb_sommets) {
    matrice_adjacence* ma = (matrice_adjacence*) malloc(sizeof(matrice_adjacence));
    ma->nb_sommets = nb_sommets;
    ma->matrice = (bool**) malloc(sizeof(bool*) * nb_sommets);
    for (sommet i = 0; i < nb_sommets; ++i) {
        ma->matrice[i] = (bool*) malloc(sizeof(bool) * nb_sommets);
        for (sommet j = 0; j < nb_sommets; ++j)
            ma->matrice[i][j] = false;
    }
    return ma;
}
```

```
void detruire_matrice_adjacence(matrice_adjacence* m) {
    if(m) {
        for (sommet i = 0; i < m->nb_sommets; ++i)
            free(m->matrice[i]);
        free(m);
    }
}
```

```
void ajouter_voisins_matrice(matrice_adjacence* m, sommet s0, sommet s1) {
    m->matrice[s0][s1] = true;
    m->matrice[s1][s0] = true;
}
```

Premier Algorithme

```
liste_triangles calcul_triangle_1(matrice_adjacence* m) {
    liste_triangles lt = creer_liste_triangles();
    for (sommet i = 0 ; i < m->nb_sommets ; ++i)
        for (sommet j = i+1 ; j < m->nb_sommets ; ++j)
            for (sommet k = j+1 ; k < m->nb_sommets ; ++k)
                if ( m->matrice[i][j] && m->matrice[j][k] && m->matrice[k][i] )
                    ajouter_triangle(&lt, creer_triangle(i,j,k));
    return lt;
}
```

`ajouter_triangle` est en $\mathcal{O}(1)$ donc l'algorithme est trivialement en $\mathcal{O}(|V|^3)$.

2 Algorithmes alternatifs

Premier Algorithme Alternatif

Algorithm 1 Calcul de $L_1 \cap L_2$

Require: L_1 et L_2 triées

1: $n_1 \leftarrow$ `longueur`(L_1), $n_2 \leftarrow$ `longueur`(L_2)
2: $L \leftarrow \emptyset$, $i_1 \leftarrow 0$, $i_2 \leftarrow 0$
3: **while** $i_1 < n_1$ et $i_2 < n_2$ **do**
4: **if** $L_1[i_1] = L_2[i_2]$ **then**
5: Ajouter $L_1[i_1]$ en tête de L
6: Incrémenter i_1 et i_2
7: **else if** $L_1[i_1] < L_2[i_2]$ **then**
8: Incrémenter i_1
9: **else** $\{L_1[i_1] > L_2[i_2]\}$
10: Incrémenter i_2
11: **end if**
12: **end while**
13: Inverser L {Si on souhaite maintenir l'ordre des sommets}
14: **return** L

La fonction `intersection` prend en paramètre deux sommets, `i < j`, du graphe reliés par une arête, cherche les sommets `k` reliés à `i` et `j` en faisant l'intersection de leur liste d'adjacence. Chaque `k` supérieur à `j` correspond à un triangle qui n'a pas déjà été ajouté à la liste. La fonction l'ajoute donc.

```
void intersection(liste_adjacence* la, liste_triangles* lt,
                  sommet i, sommet j) {
    liste_sommets li = la->voisins[i];
    liste_sommets lj = la->voisins[j];
    while( li && lj ) {
        if (li->sommet == lj->sommet) {
            if (li->sommet > j)
                ajouter_triangle(lt, creer_triangle(i,j,li->sommet));
            li = li->suivant;
            lj = lj->suivant;
        } else if (li->sommet > lj->sommet )
            lj = lj->suivant;
        else
            li = li->suivant;
    }
}
```

L'algorithme considère deux sommets `i < j` reliés ($|E|$ possibilités) et appelle ensuite la fonction `intersection` qui tourne en $\mathcal{O}(|V|)$. La complexité finale est en $\mathcal{O}(|E| \times |V|)$.

```
liste_triangles calcul_triangle_2(liste_adjacence* la) {
    liste_triangles lt = creer_liste_triangles();
    for (sommet i = 0 ; i < la->nb_sommets ; ++i)
        for (liste_sommets li = la->voisins[i] ; li ; li = li->suivant)
            if (li->sommet > i)
                intersection(la, &lt, i, li->sommet);
    return lt;
}
```

Comme $|E| \leq |V|^2$ le second algorithme est préférable. On peut passer d'une matrice d'adjacence à une liste d'adjacence en $\mathcal{O}(|V|^2)$ donc la complexité n'est pas impactée par la représentation.

Second Algorithme Alternatif

Pour être précis, la fonction `intersection` de la partie précédente a une complexité en $\mathcal{O}(\min(\Delta_i, \Delta_j))$ où Δ_i et Δ_j sont les degrés des sommets i et j respectivement.
L'algorithme précédent est donc en $\mathcal{O}(|E| \times \Delta)$. Il est donc linéaire en la taille de G dans le cas d'un degré borné, généralement pour les graphes dont le degré est petit par rapport au nombre de sommets.

3 Algorithme Heavy-Light

Par définition, si Δ_i est le degré du sommet $i \in V$ alors nous avons :

$$\begin{aligned}
2 \times |E| &= \sum_{i \in V} \Delta_i \\
2 \times |E| &= \sum_{i \in H} \Delta_i + \sum_{i \in V \setminus H} \Delta_i \\
2 \times |E| &\geq \sum_{i \in H} \Delta_i \\
2 \times |E| &\geq |H| \times \sqrt{|E|} \\
|H| &\leq 2 \times \sqrt{|E|}
\end{aligned}$$

Pour calculer les listes de voisins lourds et légers de chaque sommet on commence par calculer en $\mathcal{O}(|E|)$ la liste des sommets lourd et légers.

Algorithm 2 `LOURD_LEGER`(G) { Calcul de L_H et L_L }

$L_H \leftarrow \emptyset, L_L \leftarrow \emptyset$
for $v \in V$ **do** $\{\mathcal{O}(\sum_{v\in V} \Delta_v) = \mathcal{O}(|E|)\}$
  if `longueur`(`voisins`$(v)) < \sqrt{E}$ **then** { $\mathcal{O}(\Delta_v)$ }
    Ajouter v dans L_H
  else
    Ajouter v dans L_L
  end if
end for
return L_H, L_L

On réalise le calcul des listes de voisins lourd et léger :

Algorithm 3 `LISTE_LOURD_LEGER`(G, L_G, L_L) { Calcul de G_H et G_L }

Require: $L_H, L_L \leftarrow$ `LOURD_LEGER`(G)
$G_H \leftarrow \emptyset, G_L \leftarrow \emptyset$
for $v \in V$ **do** $\{\mathcal{O}(\sum_{v\in V} \Delta_v) = \mathcal{O}(|E|)\}$
  $G_H[v] \leftarrow$ `voisins`$(v) \cap L_H$ $\{\mathcal{O}(\Delta_v)\}$
  $G_L[v] \leftarrow$ `voisins`$(v) \cap L_L$ $\{\mathcal{O}(\Delta_v)\}$
end for
return G_H, G_L

L'algorithme final fonctionne comme suit :

Algorithm 4 Calcul des triangles de G

$L_H, L_L \leftarrow$ `LOURD_LEGER`(G) $\{\mathcal{O}(|E|)\}$
$G_H, G_L \leftarrow$ `LISTE_LOURD_LEGER`(G, L_H, L_L) $\{\mathcal{O}(|E|)\}$
$L \leftarrow \emptyset$ {Liste des triangles}
for $v \in V$ **do** {Triangles contenant au moins un sommet de H en $\mathcal{O}(|E| \times \sqrt{|E|})$}
  for $u \in \{s > v, s \in$ `voisins`$(v)\}$ **do**
    for $w \in G_H[v] \cap G_H[u]$ **do** $\{\mathcal{O}(\min(G_H[v], G_H[u]) = \mathcal{O}(|L_H|) = \mathcal{O}(\sqrt{E})\}$
      if $w > u$ **then**
        Ajouter le triangle $\{u, v, w\}$ dans L
      end if
    end for
  end for
end for
for $v \in L_L$ **do** {Triangles contenant uniquement des sommets de $V \setminus H$ en $\mathcal{O}(|E| \times \sqrt{|E|})$}
  for $u \in \{s > v, s \in G_L(v)\}$ **do**
    for $w \in G_H[v] \cap G_H[u]$ **do** $\{\mathcal{O}(\min(G_L[v], G_L[u]) = \mathcal{O}(\min(\Delta_v, \Delta_u)) = \mathcal{O}(\sqrt{E})\}$
      if $w > u$ **then**
        Ajouter le triangle $\{u, v, w\}$ dans L
      end if
    end for
  end for
end for
return L

Voici l'implémentation des algorithmes décrit précédemment.

```
void lourd_leger(liste_adjacence* la, liste_sommets* ll, liste_sommets* lh) {
    *ll = creer_liste_sommets();
    *lh = creer_liste_sommets();
    uint longueur;
    for (sommet s = 0; s < la->nb_sommets ; ++s) {
        longueur = longueur_liste_sommets(la->voisins[s]);
        if (longueur * longueur < la->nb_sommets)
            ajouter_sommet(ll, s);
        else
            ajouter_sommet(lh, s);
    }
    inverser_liste_sommets(ll);
    inverser_liste_sommets(lh);
}
```

```
void liste_lourd_leger(liste_adjacence* la, liste_sommets ll, liste_sommets lh,
                       liste_adjacence* lal, liste_adjacence* lah) {
    for (sommet s = 0; s < la->nb_sommets ; ++s) {
        liste_sommets li = la->voisins[s], lhtmp = lh, lltmp = ll;
        while( li ) {
            bool suivant_h = true, suivant_l = true;
            if ( lhtmp ) {
                suivant_h = false;
                if (li->sommet == lhtmp->sommet) {
                    ajouter_sommet(&(lah->voisins[s]), lhtmp->sommet);
                    li = li->suivant;
                    lhtmp = lhtmp->suivant;
                    continue;
                } else if (li->sommet > lhtmp->sommet )
                    lhtmp = lhtmp->suivant;
                else
                    suivant_h = true;
            }
            if ( lltmp ) {
                suivant_l = false;
                if (li->sommet == lltmp->sommet) {
                    ajouter_sommet(&(lal->voisins[s]), lltmp->sommet);
                    li = li->suivant;
                    lltmp = lltmp->suivant;
                } else if (li->sommet > lltmp->sommet )
                    lltmp = lltmp->suivant;
                else
                    suivant_l = true;
            }
            if ( suivant_l && suivant_h )
                li = li->suivant;
        }
        inverser_liste_sommets(&(lah->voisins[s]));
        inverser_liste_sommets(&(lal->voisins[s]));
    }
}
```

```
liste_triangles calcul_triangle_3(liste_adjacence* la) {
    liste_sommets ll = NULL, lh = NULL;
    liste_adjacence *lal = creer_liste_adjacence(la->nb_sommets);
    liste_adjacence *lah = creer_liste_adjacence(la->nb_sommets);
    liste_triangles lt = creer_liste_triangles();
    lourd_leger(la, &ll, &lh);
    liste_lourd_leger(la, ll, lh, lal, lah);
    for (sommet i = 0; i < la->nb_sommets; ++i)
        for (liste_sommets li = la->voisins[i] ; li ; li = li->suivant)
            if (li->sommet > i)
                intersection(lah, &lt, i, li->sommet);
    for (liste_sommets ls = ll ; ls ; ls = ls->suivant)
        for (liste_sommets li = lal->voisins[ls->sommet] ; li ; li = li->suivant)
            if (li->sommet > ls->sommet)
                intersection(lal, &lt, ls->sommet, li->sommet);
    detruire_liste_sommets(ll);
    detruire_liste_sommets(lh);
    detruire_liste_adjacence(lal);
    detruire_liste_adjacence(lah);
    return lt;
}
```

4 Affichage des résultats

Fonctions d'affichage

```
void print_triangle(triangle* t) {
    printf("(%d, %d, %d)", t->s0 + 1, t->s1 + 1, t->s2 + 1);
}
```

```
void print_liste_triangles(liste_triangles lt) {
    printf("Liste de triangles : ");
    for (; lt; lt = lt->suivant) {
        print_triangle(lt->triangle);
        if(lt->suivant)
            printf(", ");
    }
    printf("\n");
}
```

Tests

```c
#include <stdio.h>
#include <stdlib.h>
#include "calcul_de_triangles.h"

int main( int argc, char** argv ) {
    matrice_adjacence* m = creer_matrice_adjacence(5);
    liste_adjacence* l = creer_liste_adjacence(5);

    // Matrice d'adjacence
    ajouter_voisins_matrice(m, 0, 4);
    ajouter_voisins_matrice(m, 0, 3);
    ajouter_voisins_matrice(m, 0, 1);
    ajouter_voisins_matrice(m, 1, 3);
    ajouter_voisins_matrice(m, 1, 2);
    ajouter_voisins_matrice(m, 1, 4);
    ajouter_voisins_matrice(m, 2, 4);
    ajouter_voisins_matrice(m, 3, 4);
    ajouter_voisins_matrice(m, 3, 4);
    // Liste d'adjacence
    ajouter_voisins_liste(l, 0, 4);
    ajouter_voisins_liste(l, 0, 3);
    ajouter_voisins_liste(l, 0, 1);
    ajouter_voisins_liste(l, 1, 4);
    ajouter_voisins_liste(l, 1, 3);
    ajouter_voisins_liste(l, 1, 2);
    ajouter_voisins_liste(l, 2, 4);
    ajouter_voisins_liste(l, 3, 4);
    ajouter_voisins_liste(l, 3, 4);
```

```c
// Algorithme 1
liste_triangles lt = calcul_triangle_1(m);
print_liste_triangles(lt);
detruire_liste_triangles(lt);
// Algorithme 2
lt = calcul_triangle_2(l);
print_liste_triangles(lt);
detruire_liste_triangles(lt);
// Algorithme 3
lt = calcul_triangle_3(l);
print_liste_triangles(lt);
detruire_liste_triangles(lt);
```

```
    detruire_matrice_adjacence(m);
    detruire_liste_adjacence(l);
    return EXIT_SUCCESS;
}
```