TP Hashlife : notes de correction

1. La bonne grille est la grille (e).

2. Un algo naïf consiste à observer le voisinage de chacune des cases pour la mettre à jour. Ainsi, une génération demande environ $8 \times$ (nombre de cases de la grille) opérations. Pour faire une vingtaine de métagénérations, il faut encore multiplier ce nombre par $20 \times$ (la longueur d'une métagénération). On aboutit à environ :

   $$8 \times 16384 \times 16384 \times 20 \times 35328 \simeq 1.5 \times 10^{15} \text{ opérations}$$

   Sur une machine à 2GHz, donc capable de faire 2×10^9 opérations toutes les secondes, il faudrait environ 400h de calcul : ça ne va pas être possible.

3. On cite l'algorithme de Huffman et l'algorithme LZW :

   - L'algorithme de Huffman compresse en attribuant des codes dont la longueur est d'autant plus petite que la lettre compressée est fréquente dans le texte. L'attribution des codes se fait ainsi : on insère dans une file de priorité min des arbres correspondant aux lettres et dont la priorité est le nombre d'occurrences de la lettre incriminée. Tant que cette file contient plus de deux arbres, on en extrait deux, on leur donne un père commun et on réinsère l'arbre obtenu avec comme priorité la somme de celle de ses enfants. Le code d'une lettre est l'étiquettage (gauche = 0, droite = 1 par exemple) du chemin y menant dans cet arbre.
   - L'algorithme LZW compresse à la volée en maintenant à jour un dictionnaire dont les clés sont des motifs dans le texte (initialement, il contient les appariemments lettre - code) et la valeur leur code. La compression consiste à lire le texte tant que le motif lu jusqu'ici est connu, c'est-à-dire, présent dans le dictionnaire. Dès qu'il ne l'est plus, on l'ajoute au dictionnaire avec le prochain code disponible, on émet le code du motif précédent et on reprend la lecture.

   Dans le contexte qui nous occupe, c'est-à-dire un alphabet à deux lettres, l'algorithme de Huffman n'a aucun intérêt (sauf si on découpe la représentation de l'image par blocs de k bits et qu'onconsidère que les lettres sont ces blocs). L'algorithme LZW en revanche pourrait être performant car sur un alphabet à peu de lettres la probabilité d'avoir des motifs nombreux et fréquents est élevée.

4. On complète juste les champs manquants :

```
quadtree* assemble_quadtree(quadtree* no, quadtree* ne, quadtree* so, quadtree*
↪  se)
{
    quadtree* nouvel_arbre = malloc(sizeof(quadtree));

    // Question 4
    int h = hash(no->hash, ne->hash, so->hash, se->hash);

    nouvel_arbre->nb_vivantes = no->nb_vivantes + ne->nb_vivantes +
↪  so->nb_vivantes + se->nb_vivantes;
    nouvel_arbre->hash = h;
    nouvel_arbre->hauteur = no->hauteur + 1;
    // Fin question 4

    nouvel_arbre->no = no;
    nouvel_arbre->ne = ne;
    nouvel_arbre->so = so;
```

```
    nouvel_arbre->se = se;

    return nouvel_arbre;
}
```

On obtient le hash attendu pour le fichier A (88404) et 22005 pour le fichier B.

5. Le problème vient d'un dépassement de capacité. Avec des entiers représentés sur 32 bits, le plus grand entier représentable est de l'ordre de $2^{31} \simeq 2.1 \times 10^9$. Or, vu les valeurs utilisées pour `a`, `b`, `c`, `d` et `e`, le calcul du haché pourrait manipuler des entiers plus grands encore, par exemple si le haché de tous les fils vaut 99999 (la plus grande valeur possible puisqu'on réduit modulo 100000) alors `(a * no + b * ne + c * so + d * se + e)` vaut environ 2.7×10^9.

   Pour résoudre le problème, il suffit de réduire modulo 100000 au fur et à mesure des calculs :

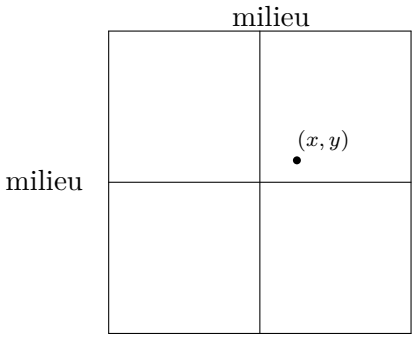
```
int hash(int no, int ne, int so, int se) {

    int a = 12421;
    int b = 6421;
    int c = 1231;
    int d = 4241;
    int e = 3499;

    // Correction apportée pour la question 5
    return ((a * no) % HASH_MAX + (b * ne) % HASH_MAX + (c * so) % HASH_MAX + (d *
↪   se) % HASH_MAX + e) % HASH_MAX;
}
```

   On trouve alors les bons hash pour C et D, 26889 pour E et 39177 pour F.

6. La couleur d'une feuille coïncide avec le fait que la cellule est vivante. Autrement, il faut détecter le quadrant dans lequel se situe le point en fonction de ses coordonnées et relancer la recherche en décalant si nécessaire lesdites coordonnées en fonction du quadrant exploré :



   Par exemple ici, x est plus petit que le milieu du côté et y est plus grand : on est donc dans le quart nord-est et il faut relancer la recherche en diminuant y de la moitié de la longueur du côté pour prendre en compte le fait qu'on ignore le quart nord-ouest.

```
int couleur_cellule(int x, int y, quadtree* t)
{
```

```
    if (t->hauteur == 0) {return t->nb_vivantes;}
    int milieu = 1 << (t->hauteur - 1);
    if (x < milieu)
    {
        if (y < milieu)
        {
            return couleur_cellule(x, y, t->no);
        }
        else
        {
            return couleur_cellule(x, y - milieu, t->so);
        }
    }
    else
    {
        if (y < milieu)
        {
            return couleur_cellule(x - milieu, y, t->ne);
        }
        else
        {
            return couleur_cellule(x - milieu, y - milieu, t->se);
        }
    }
}
```

Remarques : on constate a posteriori que l'origine des axes est en haut à gauche. Le code compagnon fournit une façon efficace de calculer 2^h : lire les commentaires !

7. Il suffit d'écrire les couleurs des pixels ligne à ligne en respectant le format PBM.

```
void exporte_quadtree(quadtree* t, char* nom)
{
    assert(t != NULL);
    assert(nom != NULL);
    int largeur_image = 1 << t->hauteur;
    FILE* flux = fopen(nom, "w");
    fprintf(flux, "P1\n%d %d\n", largeur_image, largeur_image);
    for (int i=0; i < largeur_image; i++)
    {
        for (int j=0; j < largeur_image; j++)
        {
            fprintf(flux, "%d", couleur_cellule(i, j, t));
        }
        fprintf(flux, "\n");
    }
    fclose(flux);
}
```

On peut alors observer l'image du fichier A (idem pour B ou C) en ajoutant dans le `main` :

```
char* petit_vaisseau = "data/petit_vaisseau.rle";
quadtree* pv = importe_quadtree_rle(petit_vaisseau);
exporte_quadtree(pv, "petit_vaisseau.pbm");
free_quadtree(pv);
```

Remarques : avant d'exporter les fichiers A, B, C, il faut en exporter un pour lequel l'image est fournie par l'énoncé pour pouvoir tester le code !

8. En dessinant le quadtree sur deux étages, on se rend compte que le centre peut être extrait en récupérant 4 des des sous-arbres à profondeur 2 :

```
quadtree* centre(quadtree* t)
{
    assert(t->hauteur >= 2);
    return assemble_quadtree(t->no->se, t->ne->so, t->so->ne, t->se->no);
}
```

Pour obtenir le hash du centre du fichier D, on ajoute dans le `main` :

```
char* source = "data/space_filler.rle";
quadtree* t = importe_quadtree_rle(source);
quadtree* c = centre(t);
printf("Hash du centre de la grille D : %d\n", c->hash);
free_quadtree(t);
free(c);
```

Attention, `t` et `c` partagent tous leurs noeuds sauf la racine, il faut donc bien libérer un seul des deux avec `free_quadtree` et l'autre uniquement avec `free`.

9. On procède récursivement. Note : dans le code compagnon initial, on peut se demander l'intérêt de dupliquer `allocation_feuille` en `feuille` mais cela s'éclaire lorsqu'on mémoise les noeuds.

```
quadtree* blanc(int hauteur)
{
    if (hauteur == 0) {return feuille(0); }
    quadtree* b1 = blanc(hauteur - 1);
    quadtree* b2 = blanc(hauteur - 1);
    quadtree* b3 = blanc(hauteur - 1);
    quadtree* b4 = blanc(hauteur - 1);
    return assemble_quadtree(b1, b2, b3, b4);
}
```

Le hash d'un quadtree blanc de hauteur 10 vaut 12925. On constate que le calcul puis la libération d'un tel quadtree n'est pas instantanée ce qui n'est pas très étonnant puisqu'il faut créer puis détruire $\sum_{i=0}^{9} 4^1 0 \sim 1.3 \times 10^9$ noeuds (nombre de noeuds d'un arbre d'arité 4, complet et de hauteur 10).

10. Il suffit de faire l'assemblage de 16 quadtree : pour chaque quart, il y en a trois blancs et un significatif (le même que celui déterminé pour le centre) :

```c
quadtree* entoure(quadtree* t)
{
    assert(t->hauteur > 0);
    int n = t->hauteur - 1;
    quadtree* no = assemble_quadtree(blanc(n), blanc(n), blanc(n), t->no);
    quadtree* ne = assemble_quadtree(blanc(n), blanc(n), t->ne, blanc(n));
    quadtree* so = assemble_quadtree(blanc(n), t->so, blanc(n), blanc(n));
    quadtree* se = assemble_quadtree(t->se, blanc(n), blanc(n), blanc(n));
    return assemble_quadtree(no, ne, so, se);
}
```

On pourrait aussi créer un quadtree blanc dont on modifie les cases centrales.Le sujet demande de ne pas se préoccuper de la libération de ces objets. Le hash de D entourée est de 89154.

11. Le principe d'une table de hachage est de stocker des données dans un tableau à l'aide d'une fonction dite de hachage. Le stockage d'une donnée se fait en calculant une empreinte à partir de la donnée indiquant la case où la stocker. Dans le cas où deux données ont la même empreinte (collision), la gestion peut se faire de diverses manières, par exemple par chaînage en stockant dans une liste chaînée toutes les données dont l'empreinte indique qu'elles doivent être placées dans une même case. Ce type de structure peut être utilisée pour implémenter un dictionnaire.

12. Pas de difficulté, on initialise juste tous les champs :

```c
table_hachage* nouvelle_table(int taille)
{
    table_hachage* ht = malloc(sizeof(table_hachage));
    ht->taille = taille;
    ht->nb_elems = 0;
    ht->blanc = allocation_feuille(0);
    ht->noir = allocation_feuille(1);
    ht->alveoles = malloc(HASH_MAX*sizeof(noeud*));
    for (int i = 0; i < HASH_MAX; i++)
    {
        ht->alveoles[i] = NULL;
    }
    return ht;
}
```

Remarque : l'énoncé ne donne pas le prototype de cette fonction mais il apparaît commenté dans le .h et indirectement dans le ***main****. On peut se demander l'intérêt d'avoir un argument pour cette fonction puisque de toutes façons l'énoncé dit qu'une table de hachage sera forcément de taille* ***HASH_MAX****.*

13. La difficulté est qu'il ne faut pas utiliser `free_quadtree` pour libérer les quadtrees dans les listes de la table de hachage mais uniquement le noeud racine puisque ses noeuds descendants sont dans la table de hachage aussi et donc sont eux-mêmes des racines stockées dans une autre liste dans la table (voire la même). On construit donc une fonction de libération de liste adaptée à ce contexte :

```c
void free_liste_quadtree(noeud* l)
{
    if (l != NULL)
    {
        free_liste_quadtree(l->suivant);
        free(l->t);
        free(l);
    }
}
```

Puis on libère tout dans l'ordre.

```c
void free_table_hachage(table_hachage* ht)
{
    free(ht->blanc);
    free(ht->noir);
    for (int i = 0; i < ht->taille; i++)
    {
        free_liste_quadtree(ht->alveoles[i]);
    }
    free(ht->alveoles);
    free(ht);
}
```

14. On suit les indications de l'énoncé :

```c
quadtree* feuille(int couleur)
{
    extern table_hachage* ht;
    if (couleur == 0) return ht->blanc;
    else return ht->noir;
}
```

15. Voici la nouvelle fonction d'assemblage :

```c
quadtree* assemble_quadtree(quadtree* no, quadtree* ne, quadtree* so, quadtree*
    se)
{
    extern table_hachage* ht;
    int h = hash(no->hash, ne->hash, so->hash, se->hash);

    noeud* n = ht->alveoles[h];
    while (n != NULL)
    {
        if (n->t->no == no && n->t->ne == ne && n->t->so == so && n->t->se == se)
        {
            return n->t;
        }
        n = n->suivant;
```

```
14      }
15
16      quadtree* nouvel_arbre = malloc(sizeof(quadtree));
17      nouvel_arbre->nb_vivantes = no->nb_vivantes + ne->nb_vivantes +
    ↪   so->nb_vivantes + se->nb_vivantes;
18      nouvel_arbre->hash = h;
19      nouvel_arbre->hauteur = no->hauteur + 1;
20      nouvel_arbre->no = no;
21      nouvel_arbre->ne = ne;
22      nouvel_arbre->so = so;
23      nouvel_arbre->se = se;
24
25      noeud* nouveau_noeud = malloc(sizeof(noeud));
26      nouveau_noeud->suivant = ht->alveoles[h];
27      nouveau_noeud->t = nouvel_arbre;
28      ht->alveoles[h] = nouveau_noeud;
29      ht->nb_elems = ht->nb_elems + 1;
30
31      return nouvel_arbre;
32  }
```

Le lignes 6 à 14 vérifient si le quadtree demandé existe déjà dans la table : si oui, on le renvoie sans rien allouer. Sinon, il faut le construire (lignes 16 à 23, qui correspond au code de la question 4) et le stocker dans la table (lignes 25 à 29) avant de le renvoyer.

À partir de maintenant pour les tests il suffit de créer une table en début de test et de la libérer à la fin (on peut même utiliser la même table pour plusieurs quadtrees si on n'est pas intéressée par le nombre de noeuds d'un quadtree en particulier). Il ne faut plus utiliser `free_quadtree`.

16. Sauf erreur de ma part, il n'y a rien à changer puisque les nouvelles versions de `assemble_quadtree` et `feuille` se chargent déjà de faire le travail.

    On retrouve un hash pour un quadtree blanc de hauteur 10 valant 12925 avec un calcul instantané.

17. Pour compter le nombre de noeuds réellement utilisés dans la construction d'un quadtree, il suffit d'observer le nombre d'éléments stockés dans la table de hachage après sa construction. On ajoute dans le `main` pour le fichier E (et pareil pour le F) :

```
ht = nouvelle_table(HASH_MAX);
char* course = "data/course.rle";
quadtree* c = importe_quadtree_rle(course);
printf("Nombre de noeuds fichier E : %d\n", ht->nb_elems);
free_table_hachage(ht);
```

On trouve les résultats suivants :

Fichier	Nombre de noeuds naïf	Nombre de noeuds hachés
E	196605	173
F	196605	291

Le nombre de noeuds naïf est calculé grâce à la hauteur des arbres, ici, 7. On divise par 1000 environ l'occupation mémoire !

18. Un peu long mais ce n'est qu'une traduction du dessin en faisant attention d'extraire les bons blocs à chaque étape. Les lignes 13 à 21 font la première étape de découpage en 9 blocs, les lignes 23 à 31 extraient les blocs A, B,..., les lignes 33 à 36 les combinent pour obtenir les 4 blocs qu'on fait évoluer en ligne 36 avant de les assembler.

```
1  quadtree* evolution(quadtree* t)
2  {
3
4      assert(t->hash >= 0);
5      assert(t->hash < HASH_MAX);
6
7      if (t->hauteur == 2)
8      {
9          quadtree* s = evolution_4x4(t);
10         return s;
11     }
12
13     quadtree* hg = t->no;
14     quadtree* hm = assemble_quadtree(t->no->ne, t->ne->no, t->no->se, t->ne->so);
15     quadtree* hd = t->ne;
16     quadtree* mg = assemble_quadtree(t->no->so, t->no->se, t->so->no, t->so->ne);
17     quadtree* mm = centre(t);
18     quadtree* md = assemble_quadtree(t->ne->so, t->ne->se, t->se->no, t->se->ne);
19     quadtree* bg = t->so;
20     quadtree* bm = assemble_quadtree(t->so->ne, t->se->no, t->so->se, t->se->so);
21     quadtree* bd = t->se;
22
23     quadtree* a = centre(hg);
24     quadtree* b = centre(hm);
25     quadtree* c = centre(hd);
26     quadtree* d = centre(mg);
27     quadtree* e = centre(mm);
28     quadtree* f = centre(md);
29     quadtree* g = centre(bg);
30     quadtree* h = centre(bm);
31     quadtree* i = centre(bd);
32
33     quadtree* bloc_no = assemble_quadtree(a, b, d, e);
34     quadtree* bloc_ne = assemble_quadtree(b, c, e, f);
35     quadtree* bloc_so = assemble_quadtree(d, e, g, h);
36     quadtree* bloc_se = assemble_quadtree(e, f, h, i);
37
38     quadtree* resultat = assemble_quadtree(evolution(bloc_no),
   ↪  evolution(bloc_ne), evolution(bloc_so), evolution(bloc_se));
39
40     return resultat;
41 }
```

Après avoir fait évoluer la grille D une fois puis extrait son centre, on trouve un hash valant 1253.

19. Il suffit d'itérer des évolutions en entourant la grille auparavant (en effet, l'évolution n'affecte que le

centre de la grille ; or le centre de la grille G entourée est égal à G) :

```
void simulation(quadtree* t, int nb_etapes, char* nom_dossier)
{

    assert(t != NULL);
    assert(nb_etapes > 0);
    char chemin_vers_fichier[100];
    int numero_image = 0;
    while(numero_image < nb_etapes)
    {
        snprintf(chemin_vers_fichier, sizeof(chemin_vers_fichier), "%s/%d.pbm",
↪   nom_dossier, numero_image);
        t = evolution(entoure(t));
        exporte_quadtree(t, chemin_vers_fichier);
        numero_image++;
    }
}
```

20. En faisant évoluer la course sur environ 350 étapes, on obtient le classement suivant : 3, 4, 2, 5, 1.