Coadage par Plage - Correction

Samy Jaziri

1 Compression des séquences identiques

Structure de pile

La structure de donnée est définie comme ceci :

```c
typedef unsigned int uint;

typedef struct maillon {
    char caractere;
    uint repetition;
    struct maillon* suivant;
} maillon;

typedef struct pile {
    maillon* sommet;
    uint taille;
    uint nb_maillon;
} pile;
```

On conserve un champ `taille` et un champ `nb_maillon` pour accéder à la taille de la pile et au nombre d'éléments en $\mathcal{O}(1)$. Ces champs seront mis à jour dans les fonctions `empiler` et `depiler` en temps constant.

```c
// Initialisation de pile
pile* creer_pile_vide() {
    pile* p = (pile*) malloc(sizeof(pile));
    p->sommet = NULL;
    p->taille = 0;
    p->nb_maillon = 0;
    return p;
}
```

```c
// Déstruction de la pile
void detruire_pile(pile* p) {
    maillon* n;
    while (p->sommet) {
        n = p->sommet->suivant;
        free(p->sommet);
        p->sommet = n;
    }
    free(p);
}
```

```c
void empiler(pile* p, char c) {
    maillon* m;
    if(p->sommet && p->sommet->caractere == c)
        ++p->sommet->repetition;
    else {
        m = (maillon*) malloc(sizeof(maillon));
        m->caractere = c;
        m->repetition = 0;
        m->suivant = p->sommet;
        p->sommet = m;
        ++p->nb_maillon;
    }
    ++p->taille;
}
```

```c
// Prérequis : la pile est non vide
char depiler(pile* p) {
    char c = p->sommet->caractere;
    maillon* n;
    if(p->sommet->repetition == 0) {
        n = p->sommet->suivant;
        free(p->sommet);
        p->sommet = n;
        --p->nb_maillon;
    } else
        --p->sommet->repetition;
    --p->taille;
    return c;
}
```

Codage

Le codage consiste simplement à empiler les caractères un par un.

```
pile* codage_chaine(char* s) {
    pile* encodeur = creer_pile_vide();
    for(;*s;++s)
        empiler(encodeur,*s);
    return encodeur;
}
```

`empiler` tourne en temps constant donc `codage_chaine` est en $\mathcal{O}(n)$ avec n la longueur de la chaîne à encoder.
Le décodage consiste à dépiler les éléments un per un.

```
char* decodage_chaine(pile *p) {
    char* s_decodee = (char*) malloc(sizeof(char) * p->taille + 1);
    char* tmp = s_decodee+p->taille;
    *tmp = '\0';
    while(p->sommet)
        *(--tmp) = depiler(p);
    return s_decodee;
}
```

`depiler` tourne en temps constant donc `decodage_chaine` est en $\mathcal{O}(n)$ avec n la taille de la pile. La taille de la pile est exactement la longueur de la chaîne encodée.
La pile représentant une chaîne n'occupe pas nécessairement moins d'espace que la chaîne. En effet, si la chaîne n'a pas de séquence de caractère identique, la pile contiendra autant d'élément que la chaîne. Or un élément de la pile occupe

$$sizeof(uint) + sizeof(char) + sizeof(maillon*) = 4 + 1 + 8 = 13 \text{ octets}$$

Alors qu'un caractère occupe 1 octet.
Il faut donc, pour que ce codage soit intéressant, que la chaîne soit majoritairement composée de longue séquence de caractère. Il devient intéressant de remplacer une séquence par son codage lorsqu'elle dépasse 13 caractères. Il faut par ailleurs que le gain réaliser sur chaque longue séquence, compense la perte réalisée sur les courtes séquences (moins de 13 caractères).

Liste aléatoire

On défini une fonction générant S_n.

```
char* s(uint u0, uint n) {
    uint u = u0;
    char* s = (char*) malloc(sizeof(char) * (n + 2));
    s[n+1] = '\0';
    for(uint i = 0; i <= n; i++) {
        s[i] = u % 144 == 0 ? 'N' : 'B';
        u = ( u * 15091 ) % 64007;
    }
    return s;
}
```

2 Codage par nombre optimal de carrés blancs

Type Image

```
typedef struct image {
    char** pixels;
    uint largeur;
    uint hauteur;
} image;
```

```
image* creer_image_noire(uint largeur, uint hauteur) {
    image* img = (image*) malloc(sizeof(image));
    img->largeur = largeur;
    img->hauteur = hauteur;
    img->pixels = (char**) malloc(sizeof(char*) * largeur);
    for(uint i = 0 ; i < img->largeur ; ++i) {
        img->pixels[i] = (char*) malloc(sizeof(char) * img->hauteur);
        for(uint j = 0 ; j < img->hauteur ; ++j)
            img->pixels[i][j] = 'N';
    }
    return img;
}
```

```
void detruire_image(image* img) {
    for(uint i = 0 ; i < img->largeur ; ++i)
        free(img->pixels[i]);
    free(img->pixels);
    free(img);
}
```

Plus grand carré blanc

Notons `PGCB(x,y)` le plus grand carré blanc dont le coin inférieur droit est le pixel (x, y).
Pour résoudre le sous-problème $(\star)$ associé au pixel (x, y) on remarque qu'un carré blanc dont le coin inférieur droit est (x, y) de taille n est égal à l'union des carrés blancs de taille $n-1$ dont les coins inférieurs droit sont $(x-1, y), (x, y-1) et (x-1, y-1)$ et de (x, y) lui-même.
On montre que `PGCB(x,y) = 1 + min(PGCB(x-1,y), PGCB(x,y-1), PGCB(x-1,y-1))`.
Soit n la taille de `PGCB(x,y)`. On peut en conclure que les trois carrés de taille $n-1$ dont les coins inférieur droit sont $(x-1, y), (x, y-1) et (x-1, y-1)$ sont blancs et inclus dans `PGCM(x,y)`. Au moins l'un de ces carrés est maximal, sinon les pixels $(x-n, y), (x-n, y-1), \ldots, (x-n, y-n)$ appartiendrait à `PGCB(x-1,y)` et les pixels $(x, y-n), (x-1, y-n), \ldots, (x-n, y-n)$ appartiendrait à `PGCB(x,y-1)`. Ces pixels serait donc blanc et on pourrait étendre `PGCB(x,y)` pour qu'il forme un carré blanc de taille $n+1$ dont le coin inférieur est (x, y). Ce contredirait la maximalité de `PGCB(x,y)`.
A l'inverse, si on prend `n = 1 + min(PGCB(x-1,y), PGCB(x,y-1), PGCB(x-1,y-1))` on note déjà que le carré de taille n et aillant comme coin inférieur droit le pixel (x, y) est un carré blanc. Si il n'était pas maximal, le carré de taille $n+1$ et aillant comme coin inférieur droit le pixel (x, y) serait un carré blanc. Les trois carrés de taille n dont les coins inférieur droit sont $(x-1, y), (x, y-1) et (x-1, y-1)$ serait blancs ce qui contredirait l'hypothèse de maximalité d'un des carrés `PGCB(x-1,y)`, `PGCB(x,y-1)`, ou `PGCB(x-1,y-1)`.

```
uint min3( uint a, uint b, uint c ) {
    if(a < b && a < c)
        return a;
    if(b < c)
        return b;
    return c;
}
```

```
uint plus_grand_carre_blanc(image* img) {
    uint max = 0;
    uint** matrice = (uint**) malloc(sizeof(uint*) * img->largeur);
    for(uint i = 0 ; i < img->largeur ; ++i) {
        matrice[i] = (uint*) malloc(sizeof(uint) * img->hauteur);
        for(uint j = 0 ; j < img->hauteur ; ++j)
            if(img->pixels[i][j] == 'B') {
                matrice[i][j] = 1;
                if(i > 0 && j > 0)
                    matrice[i][j] += min3(matrice[i-1][j],
                                          matrice[i][j-1],
                                          matrice[i-1][j-1]);
                if(matrice[i][j] > max)
                    max = matrice[i][j];
            } else {
                matrice[i][j] = 0;
            }
    }
    for(uint i = 0 ; i < img->largeur ; ++i)
        free(matrice[i]);
    free(matrice);
    return max;
}
```

L'algorithme utilise un tableau de dimension $l \times h$ et produit une image de dimensions $l \times h$, la complexité est donc en $\mathcal{O}(l\,h)$.

Image noir et blanc aléatoires

On définit une fonction `i` qui renvoie l'image associée à la chaîne S_n :

```
image* i(char* s, uint t) {
    image* img = creer_image_noire(t,t);
    for(uint i = 0 ; i < img->largeur ; ++i)
        for(uint j = 0 ; j < img->largeur ; ++j)
            img->pixels[i][j] = s[i * 212 + j];
    return img;
}
```

Codage

Lors du calcul de `PGCB(x,y)` on décide de garder `PGCB(x,y)` et d'éliminer parmi `PGCB(x-1,y)`, `PGCB(x,y-1)`, et `PGCB(x-1,y-1))` les carrés entièrement inclus dans `PGCB(x,y)` et de garder les autres. On ne peut pas obtenir un plus petit nombre de carrés couvrant sans modifier les carrés déjà calculer et donc s'éloigner d'une heuristique gloutonne. On conservera la liste des carrés à

conserver dans une matrice de booléen représentant pour chaque pixel, est-ce que `PGCB(x,y)` sera conservé dans le codage final ou non.

```
uint* codage_image(image* img) {
    uint taille_code = 0;
    uint *code, *code_tmp;
    bool** coins = (bool**) malloc(sizeof(bool*) * img->largeur);
    uint** matrice = (uint**) malloc(sizeof(uint*) * img->largeur);
    for(uint i = 0 ; i < img->largeur ; ++i) {
        coins[i] = (bool*) malloc(sizeof(bool) * img->hauteur);
        matrice[i] = (uint*) malloc(sizeof(uint) * img->hauteur);
        for(uint j = 0 ; j < img->hauteur ; ++j) {
            if(img->pixels[i][j] == 'B') {
                matrice[i][j] = 1;
                coins[i][j] = true;
                ++taille_code;
                if(i > 0 && j > 0) {
                    matrice[i][j] += min3(matrice[i-1][j],
                                          matrice[i][j-1],
                                          matrice[i-1][j-1]);
                    if(coins[i-1][j] && matrice[i][j] > matrice[i-1][j]) {
                        coins[i-1][j] = false;
                        --taille_code;
                    }
                    if(coins[i][j-1] && matrice[i][j] > matrice[i][j-1]) {
                        coins[i][j-1] = false;
                        --taille_code;
                    }
                    if(coins[i-1][j-1] && matrice[i][j] > matrice[i-1][j-1]) {
                        coins[i-1][j-1] = false;
                        --taille_code;
                    }
                }
            } else {
                matrice[i][j] = 0;
                coins[i][j] = false;
            }
        }
    }
```

```
    code = (uint*) malloc(sizeof(uint) * ( taille_code * 3 + 3) );
    code_tmp = code;
    *code_tmp = taille_code;
    *(++code_tmp) = img->largeur;
    *(++code_tmp) = img->hauteur;
    for(uint i = 0 ; i < img->largeur ; ++i) {
        for(uint j = 0 ; j < img->hauteur ; ++j) {
            if(coins[i][j]) {
                *(++code_tmp) = i;
                *(++code_tmp) = j;
                *(++code_tmp) = matrice[i][j];
            }
        }
    }
    for(uint i = 0 ; i < img->largeur ; ++i) {
        free(matrice[i]);
        free(coins[i]);
    }
    free(matrice);
    free(coins);
    return code;
}
```

L'algorithme tourne en $\mathcal{O}(n^2)$ et utilise deux tableaux de dimension n^2 en plus de renvoyer une image de dimension $n \times n$. La complexité spatiale est donc $\mathcal{O}(n^2)$.

```
image* decodage_image(uint* code) {
    uint taille_code = code[0];
    uint largeur = code[1];
    uint hauteur = code[2];
    uint x, y, c;
    image* img = creer_image_noire(largeur,hauteur);
    for(uint k = 3 ; k <= taille_code * 3 ; k += 3) {
        x = code[k];
        y = code[k+1];
        c = code[k+2];
        for(uint i = 0; i < c ; ++i)
            for(uint j = 0; j < c ; ++j)
                img->pixels[x-i][y-j] = 'B';
    }
    return img;
}
```

3 Affichage des résultats de la fiche réponse

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

int main( int argc, char** argv ) {
    if (argc != 2) {
        fprintf(stderr, "Usage : %s u0\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    int u0 = atoi(argv[1]);
    char* s1000 = s(u0, 1000);
    char* s100000 = s(u0, 100000);
    char* s12000 = s(u0, 12000);
    pile* code_s1000 = codage_chaine(s1000);
    pile* code_s12000 = codage_chaine(s12000);
    pile* code_s100000 = codage_chaine(s100000);
    printf("Question 4 : a) %i  b) %i  c) %i\n",
           code_s1000->nb_maillon,
           code_s12000->nb_maillon,
           code_s100000->nb_maillon);

    char* s212x212 = s(u0, 212*212);
    char* s512x512 = s(u0, 512*512);
    char* s1000x1000 = s(u0, 1000*1000);
    image* img212x212 = i(s212x212,212);
    image* img512x512 = i(s512x512,512);
    image* img1000x1000 = i(s1000x1000,1000);
    printf("Question 8 : a) %i  b) %i  c) %i\n",
           plus_grand_carre_blanc(img212x212),
           plus_grand_carre_blanc(img512x512),
           plus_grand_carre_blanc(img1000x1000));
    uint* code212x212 = codage_image(img212x212);
    uint* code512x512 = codage_image(img512x512);
    uint* code1000x1000 = codage_image(img1000x1000);
    printf("Question 11 : a) %ld%%  b) %ld%%  c) %ld\%%\n",
           100 - ((code212x212[0]*3+3) * sizeof(uint) * 100)
            / (212*212*sizeof(char)) ,
           100 - ((code512x512[0]*3+3) * sizeof(uint) * 100)
            / (512*512*sizeof(char)) ,
           100 - ((code1000x1000[0]*3+3) * sizeof(uint) * 100)
            / (1000*1000*sizeof(char)) );
```

```
    free(s1000);
    free(s100000);
    free(s12000);
    detruire_pile(code_s1000);
    detruire_pile(code_s12000);
    detruire_pile(code_s100000);
    free(s212x212);
    free(s512x512);
    free(s1000x1000);
    detruire_image(img212x212);
    detruire_image(img512x512);
    detruire_image(img1000x1000);
    free(code212x212);
    free(code512x512);
    free(code1000x1000);

    return EXIT_SUCCESS;
}
```