Rapport de projet, APA

Vincent JOULAIN & Badr OUARAFANA & Remy GARDETTE

Decembre 2022

Table des matières

1 Experimentation des algorithmes fournis

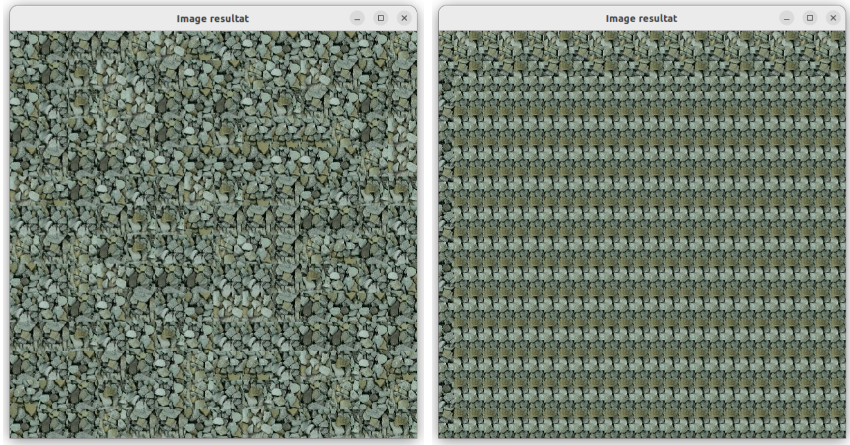


Figure 1 – algo a et b sur l'image gravier.tif

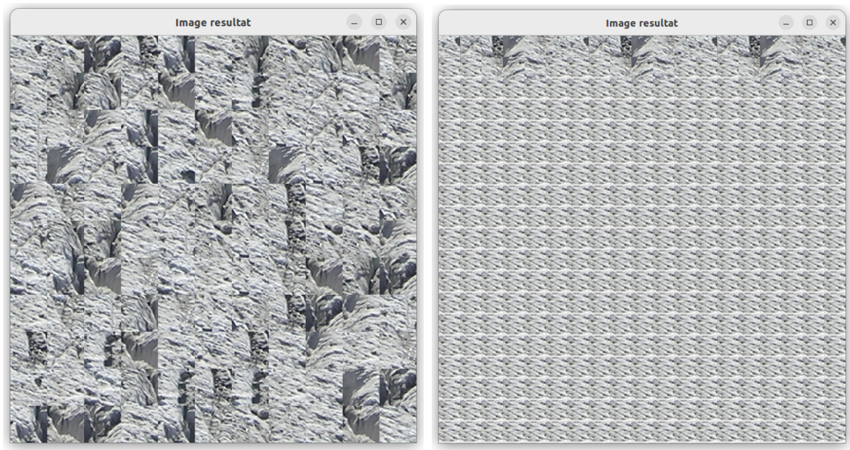


Figure 2 – algo a et b avec 25 blocs (5 en paramètre) sur l'image glace66.tif

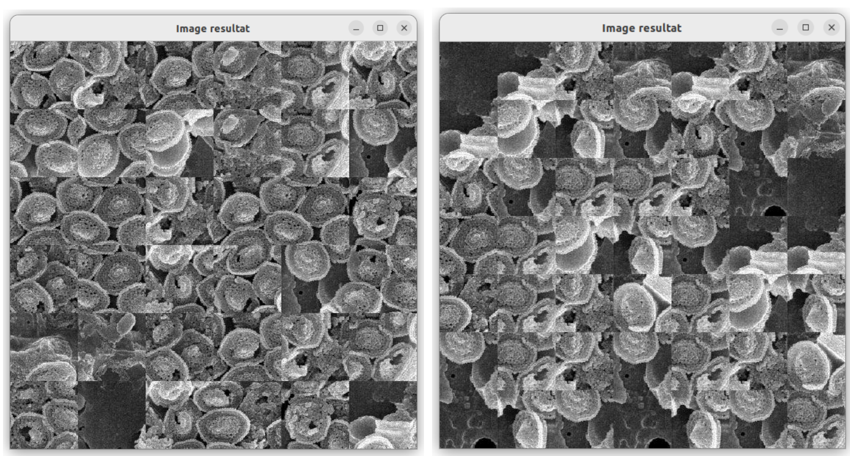


Figure 3 – algo a et b avec 25 blocs (5 en paramètre) sur l'image test4.tif

2 Algorithme générateur aléatoire

```
Permuteur::Permuteur(int max){
	this->max=max;
	this->indices = (int*)malloc(max*sizeof(int));
	this->perm = (int*)malloc(max*sizeof(int));
	this->i_perm=0;

	for (int i =0 ; i < max ; i++){
		indices[i]=i;
		perm[i]=i;
	}

	for (int i=0; i<max; i++) {
		int r=rand()%(max-i);
		int c=perm[r];
		perm[r]=perm[i];
		perm[i]=c;
	}
}

int Permuteur::suivant(){
	if(this->i_perm==this->max){
	for (int i=0; i<this->max; i++) {
		int r=rand()%(this->max-i);
		int c=this->perm[r];
		this->perm[r]=this->perm[i];
		this->perm[i]=c;
	}
	this->i_perm=0;
	}
	return this->perm[this->i_perm++];
}

Permuteur::~Permuteur(){
	free(this->indices);
	free(this->perm);
}
```

Cet algorithme va générer une permutation aléatoire d'un tableau de taille "max", contenant les valeurs de 0 à max-1. À chaque appel de la fonction suivant() on retourne une case du tableau. Quand on atteint la dernière case, on relance une permutation aléatoire.

3 Réprésentation de e et E

e_{ij} représente la distance entre deux pixels d'un block pour chaque pixel dans le block de recouvrement, (l'erreur du pixel dans la zone de recouvrement en position i, j).

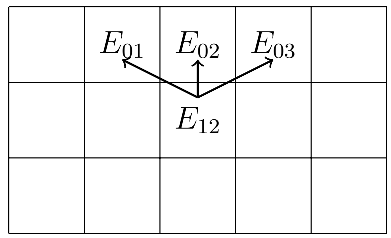
E est la somme des e_{ij} tel que : $E = \sum e_{ij}$ est une solution optimale.

Trouver la solution optimale revient à trouver le minimum de l'ensemble des chemins $E_0, E_1 \cdots E_n$

4 Rapport entre le coût de la coupe optimale et les E_{ij}

On cherche à obtenir une coupe optimale verticalement, qui peut être calculé récursivement, ou itérativement.

$$E_{ij} = \begin{cases} e_{ij} & i = 0 \\ e_{ij} + min(E_{i-1j-1}, E_{i-1j}, E_{i-1j+1}) & i > 0 \end{cases}$$



La valeur minimale de $E = \sum e_{ij}$ est le $E_{min} = min(E_{n0}, E_{n1}, \cdots, E_{nn})$ tq. n = nombre de lignes.

La coupe minimale $CP = \{E_{i_{n-1}j_{k1}}\, E_{i_{n-2}j_{k2}} \cdots E_{i0j_{km}}\}$ est tracée à partir de $E_{min} = E_{i_{n-1}j_{k1}}$ en suivant le chemin des indices i, j tel que $i \in \{0, 1 \cdots, L-1\}, j \in \{0, 1, \cdots C-1\}$ L = lignes et C = colonnes tel que $j_k =$ indiceMin$(E_{i-1,j_k-1}, E_{i-1,j_k}, E_{i-1,j_k+1})$

5 Notion de coupe optimale avec un exemple

$$e = \begin{pmatrix} 1 & 3 & 2 & 1 \\ 2 & 1 & 2 & 3 \\ 1 & 3 & 4 & 2 \\ 2 & 4 & 3 & 1 \\ 4 & 3 & 1 & 2 \end{pmatrix} \qquad E = \begin{pmatrix} 1 & 3 & 2 & 1 \\ 3 & 2 & 3 & 4 \\ 3 & 5 & 6 & 5 \\ 5 & 7 & 8 & 6 \\ 9 & 8 & 7 & 8 \end{pmatrix}$$

On commence par calculer le tableau E à l'aide du tableau e ainsi que des précédents indices que nous avons calculés (afin d'éviter la redondance des calculs). Une fois le tableau E obtenu, on cherche l'élément le plus petit sur la dernière ligne et on remonte en cherchant les minimums (avec une amplitude de 3) afin de tracer la coupe optimale (ici 7-6-5-3-1).

6 complexité d'une solution récursive naive pour le calcul de la coupe optimale

Supposons une matrice e sans bordures et $p, n \in N$ respectivement la largeur et la hauteur de E, le cout du calcul de chacune des colonnes de E s'éleve à :

pour i de 0 à n-1	Nombre d'opérations
— — — — — —	— — — — — —
i = 0	Ø
i = 1	3
i = 2	$3^2 + 3$
i = 3	$3^3 + 3^2 + 3$
...	...
i = n-1	$3^{n-1} + 3^{n-2} + 3^{n-3} \cdots + 3^1$

supposons maintenant que e possède des bordures et calculons le cout du calcul d'une des colonne sur un des bords :

pour i de 0 à n-1	Nombre d'opérations
— — — — — —	— — — — — —
i = 0	Ø
i = 1	2
i = 2	$(2 + 3^1) + 2$
i = 3	$(2 + 3^2) + (2 + 3^1) + 2$
...	...
i = n-1	$(2 + 3^{n-2}) + (2 + 3^{n-3}) \cdots + (2 + 3^1) + 2$

Si nous souhaitions être parfaitement rigoureux, il nous faudrait aussi décompter les opérations faites lorsque une colonne centrale (qui n'est pas une bordure) atteint une bordure (droite, gauche ou les deux). Toutefois on remarque que même en prenant le calcul de la colonne la moins couteuse (c'est-à-dire $j = 0$ ou $j = p - 1$) on obtient toujours un temps exponentiel. Nous pouvons donc admettre que ces opérations que nous devrions soustraire de notre calcul représentent un temps dérisoire par rapport au calcul total.

La complexité du calcul récursif naif est $O((2\sum_{i=1}^{n-2} 2 + 3^i) + 2) + O(p - 2\sum_{i=1}^{n-1} 3^i)$ soit un **temps exponentiel**

7 Algorithme récursif naif

```
int RaccordeurRecursifNaif::calculerCoupeAlgo(MatInt2 *e, int i, int j){
    int valeur = e->get(i , j);
    if (i == 0)
        return valeur;
    else if (j <= 0)
        return valeur + std::min(calculerCoupeAlgo(e, i - 1, j), calculerCoupeAlgo(e, i - 1, j + 1));
    else if (j >= e->nColonnes() - 1)
        return valeur + std::min(calculerCoupeAlgo(e, i - 1, j - 1), calculerCoupeAlgo(e, i - 1, j));
    else
        return valeur + std::min(calculerCoupeAlgo(e, i - 1, j - 1),
        std::min(calculerCoupeAlgo(e, i - 1, j), calculerCoupeAlgo(e, i - 1, j + 1)));
}
  int RaccordeurRecursifNaif::calculerRaccord(MatInt2 *e,  int *coupe){
    int coupeMini = 0 ; // variable qui sera la valeur Emin
    int **E = new int *[e->nLignes()];  //allocation mémoire du tableau E
    for (int i = 0; i < e->nLignes(); i++){
        E[i] = new int[e->nColonnes()];
    }
    for (int i=0 ; i< e->nLignes() ; i++){
        for(int j=0 ; j< e->nColonnes() ; j++) {
            E[j][i] = calculerCoupeAlgo(e, i , j) ;  //remplissage des valeurs E ij
        }
    }
    int indiceMin = 0;

    // trouver l'indice min de la dernière ligne
    for (int j = 0; j < e->nColonnes() ; j++){
        if (E[e->nLignes() - 1][j] < E[e->nLignes() - 1][indiceMin])
            indiceMin = j;
    }
    coupe[e->nLignes() - 1] = indiceMin;
    coupeMini = E[e->nLignes() - 1][indiceMin];

    // on cherche la coupe optimale on se deplacant en diagonale à droite, à gauche ou au dessus
    for (int i = e->nLignes() - 2; i >= 0; i--){
        int start = 0;
        int end = 0;
        if (indiceMin == 0){
            start = 0;
            end = indiceMin + 1;
        }
        else if (indiceMin == e->nColonnes()  - 1){
            start = indiceMin - 1;
            end = e->nColonnes()  - 1;
        }
        else{
            start = indiceMin - 1;
            end = indiceMin + 1;
        }

        indiceMin = start;

        // On cherche l'indice min de j-1, j et j+1 de chaque valeur au dessus
        for (int x = start; x <= end; x++){
            if (E[i][x] < E[i][indiceMin]){
                indiceMin = x;
            }
        }
        coupe[i] = indiceMin ;
    }
```

```
    for (int i=0 ; i< e->nLignes()  ; i++) {
        free(E[i] );
    }
    free(E) ;
    return coupeMini ;
}
```

On utilise deux fonctions, l'algorithme récursif ainsi que le programme maitre. ce dernier s'occupe de faire appel à l'algorithme afin de remplir notre tableau E. il ne nous reste alors plus qu'a chercher la coupe optimale dans notre tableau.

L'algorithme récursif reçoit la matrice e et deux indices i et j. Afin de calculer E_{ij} on fait appel à notre même fonction pour trouver les valeurs $(E_{i-1,j-1}, E_{i-1,j}, E_{i-1,j+1})$ jusqu'a arriver sur $i = 0$.

8 Algorithme récursif optimisé

```
int RaccordeurRecursifNaifOptimise::calculerCoupeAlgo(int **tab, MatInt2 *e, int i, int j)
{
    if (tab[i][j] == -1){
        int valeur = e->get(i, j);
        if (i == 0)
            tab[i][j] = valeur;
        else if (j <= 0)
            tab[i][j] = valeur + std::min(calculerCoupeAlgo(tab, e, i - 1, j), calculerCoupeAlgo(tab, e, i - 1, j + 1));
        else if (j >= e->nColonnes() - 1)
            tab[i][j] = valeur + std::min(calculerCoupeAlgo(tab, e, i - 1, j - 1), calculerCoupeAlgo(tab, e, i - 1, j));
        else
            tab[i][j] = valeur + std::min(calculerCoupeAlgo(tab, e, i - 1, j - 1),
            std::min(calculerCoupeAlgo(tab, e, i - 1, j), calculerCoupeAlgo(tab, e, i - 1, j + 1)));
    }

    return tab[i][j];
}

```

Dans cette version, on cherche à éliminer les calculs redondants. Pour celà, on stocke les résultats obtenus dans un tableau de taille $M \times N$. Nous n'avons plus qu'a nous réferrer à ce tableau lors d'un éventuel appel à une des valeurs précedemment calculées.

9 Algorithme itératif

```
void RaccordeurIteratif::calculerCoupeAlgo(MatInt2 *e, int **E){
    int lignes = e->nLignes();
    int colonnes = e->nColonnes();

    //cas de base : pour i=0 on remplit la 1ere ligne de E avec les elements de la 1ere ligne de e
    for (int j = 0; j < colonnes; j++){
        E[0][j] = e->get(0,j);
    }
    //pour chaque ligne i+1 , on calcule Eij on se basant sur les elements de precedents [i-1,j-1], [i-1,j],[i-1,j+1]
    for (int i = 1; i < lignes; i++){
        for (int j = 0; j < colonnes; j++){
            if (j <= 0){
                E[i][j] = e->get(i,j) + std::min(E[i - 1][j], E[i - 1][j + 1]);
            }
            else if (j >= colonnes - 1) {
                E[i][j] = e->get(i,j) + std::min(E[i - 1][j - 1], E[i - 1][j]);
            }
            else{
                E[i][j] = e->get(i,j) + std::min(E[i - 1][j - 1], std::min(E[i - 1][j], E[i - 1][j + 1]));
            }
        }
    }
}
```

On commence par calculer notre cas basique, c'est-à-dire quand i vaut 0, on calcule ensuite chacune des lignes suivantes avec une boucle en deux dimensions ($lignes \times colonnes$) l'algorithme permettant de trouver la coupe optimale est identique pour les trois versions.

10 Evaluation de complexité

Comme nous l'avons vu lors de la question 6, l'agorithme (c) récursif naif possède une complexité exponentielle. Sur l'algorithme résuctif optimisé, on élimine la redondance. Par conséquent il ne nous reste alors qu'a stocker les valeurs dans le tableau E de taille $M \times N$ Sur l'algorithme itératif, nous parcourons tous les éléments du tableau E toujours de taille $M \times N$ étant donné que chaque ligne du tableau dépend de la ligne précédente. Nous pouvons conclure que la complexité des deux derniers algorithmes est $O(MN)$, soit un temps bien plus interessant que notre première version récursive naïve.

Nous avons pu remarquer au cours de nos différents essais que certaines images fonctionnaient très mal avec la synthèse de textures peu importe l'algorithme, comme par exemple l'image 13.jpg.tif.