TRAITEMENT D'IMAGES «OUT-OF-CORE»

PROJET R25

April 18, 2019

Carneiro Espíndola Stela
Durrmeyer Alexandre
Neghnagh-Chenavas Jules
Mailharro Erwan
Paris Édouard
Paulin Florian
Pov Cécile
Tuteur : Cousty Jean

Contents

Chapter 1

Introduction

Ce rapport de projet est le fruit d'un travail de recherche réalisé dans le cadre de l'unité PRJ-4000, projet annuel durant notre 4ème année à ESIEE Paris. Une demi-journée (les lundis après-midi) était consacrée à la réalisation de ce projet de recherche, dirigé par M. Jean Cousty.

1.1 Motivation

Recent imaging systems, such as electron microscopes, tomography, or medical scans used for example in research laboratories, biological development or material sciences, can produce 2D/3D large data volumes that can reach several dozens of GB. They are produced by the merging many sub-images made with a very precised zoom and

Those images are intended for analysis and quantification using image processing techniques. However, some of those algorithms, and particularly image segmentation algorithms such as watershed, were not designed for huge volumes data (huge images), that is to say data that cannot be stored into a computer's main memory at one time. Even if most of algorithms are compatible with a block decomposition approach, it is necessary for the others to fundamentally reconsider the algorithm approach.

This is particularly the case of image segmentation algorithms, including watershed segmentation algorithm. The appropriate approach for huge volumes data is called "out-of-

core".

1.2 Goal of the study

Goals of our study are the following :

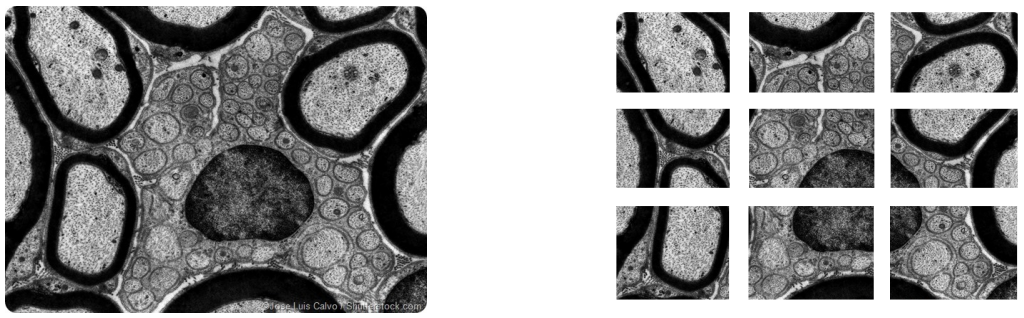
- Access the state-of-the-art of global research on watershed algorithm variants and strategies used to process large data on other image analysis problems ;
- Design a new watershed segmentation algorithm, with an «out-of-core»/external memory approach, so that we can process data that is too large to fit into a computer's memory at one time. This algorithm must produce exactly the same result as if we had processed it without dividing the blocks (in core approach).
- Produce experimental results that shows or not the relevancy of the method chosen.

Chapter 2

Method - Theory, Design, Model

2.1 Segmentation by block

In order to deal with images of very large sizes, we propose to dived the original image into smaller blocks B, that can be loadded into the memory. Figure 2.1 shows an exemple of an medical image which was dived into blocks. This image was not used in the experiments and it is just an example of how the division of the future input images can be read by our algorithm.



(a) Example image

(b) Image divided into blocks

Figure 2.1: Division of the original image into blocks

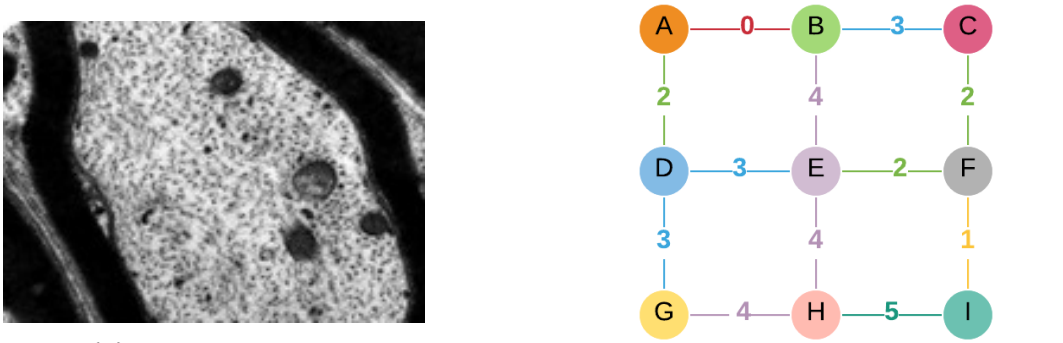
The number of blocks will be determined by the width and the hight chosen for the block. We'll call in this repport the width of the block as block_x and the hight block_y, opposed to the width and hight of the image which we'll call img_x and img_y respectively. We will name each block from left to right and then from the top to the bottom (Figure

2.2).

A	B	C
D	E	F
G	H	I

Figure 2.2: Enumaration of blocks

Each block $b \in B$ will be treated as an individual image, which will be read by our algorithm as a graph. The graph $G_b(N, E)$ it's composed by a set of nodes N and of edges E, in which each egde e $\in E$ links only to nodes of N. Each x $\in$ N represents a pixel on the original image, and each node is connected with its four neighbours. Figure 2.20 shows how the graph is built in the memory.



(a) Original block

(b) Graph G_b of the exemple block

Figure 2.3: Generation of the graph

After that we start the process of segmentation by block using the algorithm watershed.

2.2 Ligne de partage des eaux

Si l'on fait tomber une goutte d'eau sur le flac d'une montagne, celle-ci ruissellera jusqu'un potentiel « bassin de rétention ». Si l'on fait ruisseler une autre goutte d'eau sur l'autre flanc de la même montagne, celle-ci rejoindra un autre bassin de rétention. Topographiquement, une ligne de partage des eaux est la ligne séparera les deux flancs. Ainsi si l'on dépose la goutte d'eau d'un côté ou de l'autre de la ligne de partage des eaux, la goutte atteindra un bassin de rétention différent.

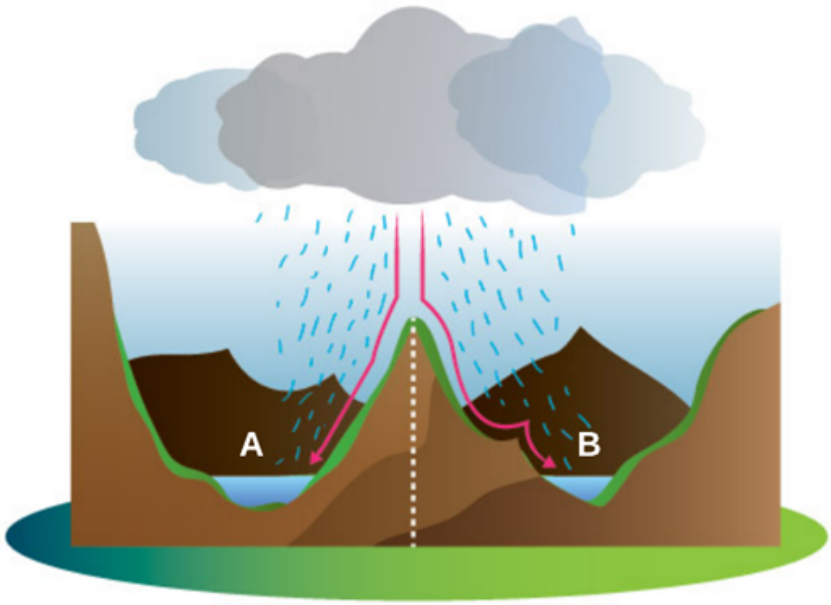


Figure 2.4: Représentation topologique d'une LPE

Cette notion de ligne de partage des eaux est intéressante dans le domaine du traitement d'image car elle peut permettre de « segmenter » une image, c'est-à-dire de séparer l'image en plusieurs régions logiquement reliées.

Tout d'abord, nous allons travailler avec des « edge weighted graphs », c'est-à-dire que chaque arc possèdera un poids. Mais comment le determiner, puisqu'une image est

une suite de pixels, chaque pixel correspondant à un sommet ? Eh bien c'est a notre bon vouloir, en fonction de ce que l'on veut faire sur l'image. Si l'on travaille sur une image en deux dimension (en noir et blanc), un arc peut être simplement la différence entre la valeurs des deux pixels qu'il relie. Si l'on travaille sur une image en couleur, il y aura probablement une autre technique à définir, etc. Pour le moment, nous travaillons avec des images en niveau de gris, et chaque arc correspond à la différence de valeur des sommets (correspondant aux pixels) qu'il joint.

Pour déterminer ce qu'est une ligne de partage des eaux algorithmiquement, expliquons les deux notions suivantes :

Soit F un graphe.

- Un **minimum de F** est un sous-graphe connexe X de F dans lequel tous les arcs ont un poids égal k (réel positif), et pour lequel tous les arcs adjacents de X sont supérieurs à k.
- Les **minima d'un graphe F**(noté M(F)) est un graphe unissant tous les minimums de F.

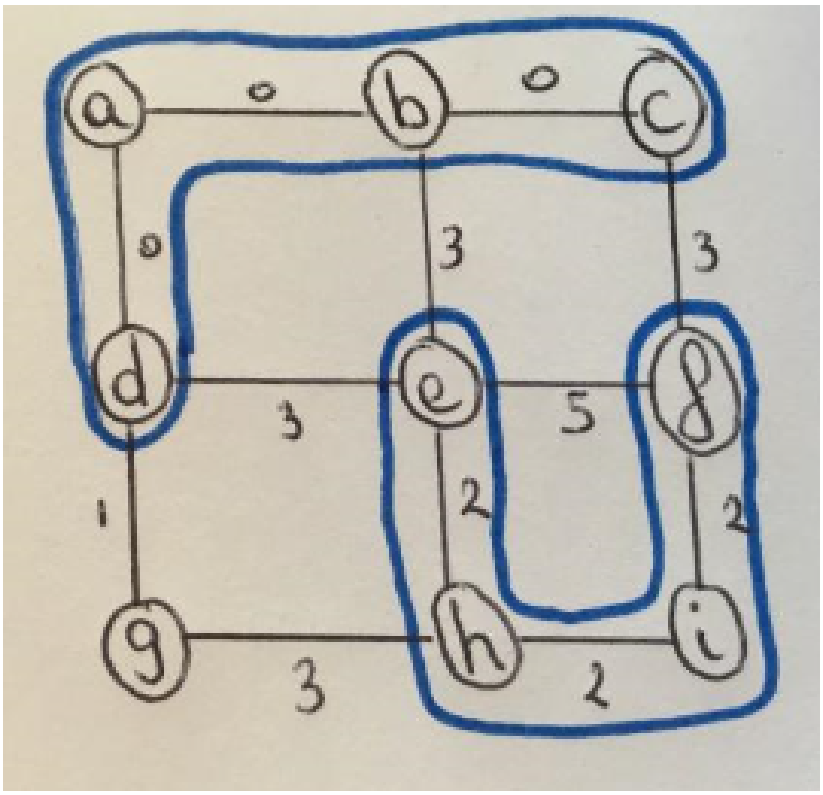


Figure 2.5: Exemple : En bleu, les minimas de F

Un **«chemin descendant»** est une suite d'arcs reliés entre eux (deux arcs sont reliés entre eux s'ils possèdent un sommet en commun) dont le poids est décroissant.

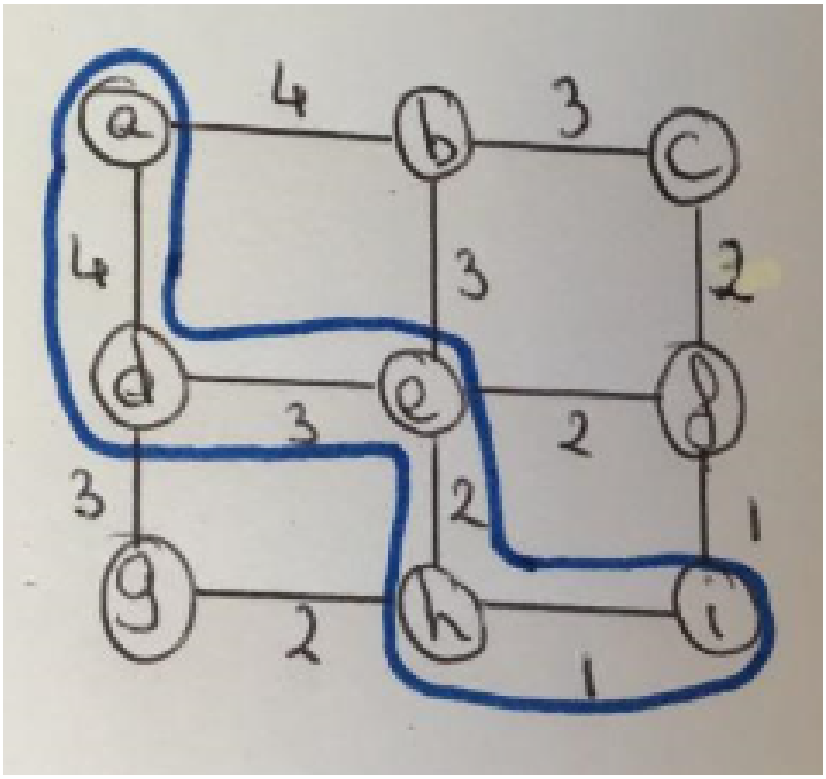


Figure 2.6: Exemple : En bleu, un chemin descendant de a à i. Il n'existe pas de chemin descendant de i à a.

Dans un «edge-weighted graph», (graphe dans lequel les arcs ont un poids (ou une altitude), une ligne des partage des eaux est une suite d'arcs de telle sorte à ce qu'on puisse trouver de part et d'autre deux chemins descendants, menant à deux minimum locaux de F distincts.

Comment peut-on alors déterminer algorithmiquement ces lignes de partage des eaux (ou « watershed cuts ») ?

Notion d'extension d'un graphe : On dit que « Y est une extension de X » si X est inclus dansY, si Y est connexe, et si tous les sommets et tous les arcs de X sont aussi dans Y.

Si l'on crée des extensions des minimums de F, de telle sorte à ce que chaque sommet de F soient dans une extension d'un des minimums de F un arc reliant deux sommets appartenant à deux extensions différentes est un arc de « coupure ». Intuitivement, il

existe au moins une « combinaisons d'extension » de sorte à ce que ces coupures soient des « watershed cuts ».

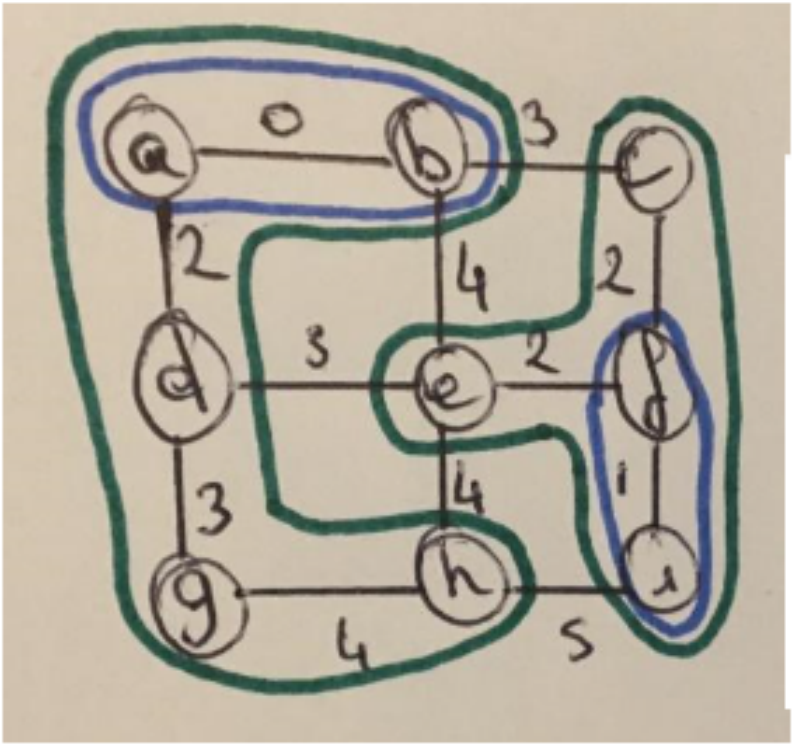


Figure 2.7: Dans cet exemple, en bleu sont entourés les minimas et en vert une extension possible des deux minimums tels que chaque sommet appartiennent à une extension. On remarque que la coupure est un watershed.

Notion de Foret : En théorie, une forêt est un graphe comportant plusieurs arbres (qui ne peuvent pas contenir de cycle). Cependant, comme les minimas de notre graphe F peuvent eux-même contenir des cycles, on dit qu'une forêt est l'union de plusieurs sous-graphes de F connexe dans lequel seuls les minimas de F peuvent contenir des cycles.

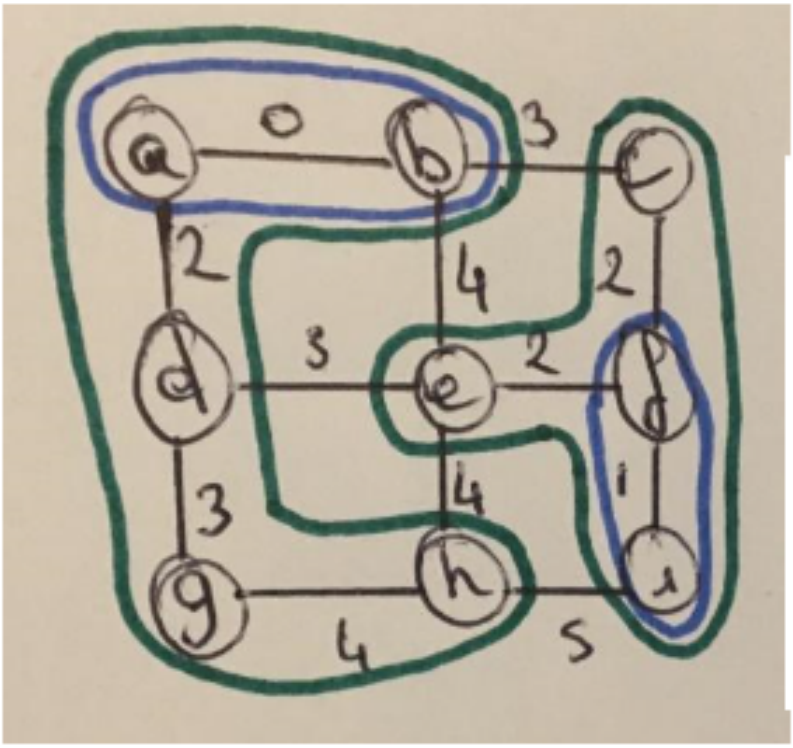


Figure 2.8: Dans cet exemple, on remarque que cette combinaison d'extension est non seulement une MSF, mais également que la coupure correspond a une watershed cut.

Ainsi, une MSF (pour « minimum spanning forest ») relative aux minimums est la forêt dont la somme des poids des arcs la constituant est la plus petite, et dans laquelle chaque arbre est une extension d'un minima.
Dans l'article, on démontre que si l'on trouve la coupure d'un MSF, celle-ci est elle-même une ligne de partage des eaux. Trouver une ligne de partage des eaux revient donc à trouver une MSF.

Pour trouver cette MSF, il y a trois méthodes. Chaque méthode sera utilisée pour un cas différent. La première peut être utilisée dans le cas de calculs « en parallèle », la seconde ne peut être exécutée en parallèle, mais est en complexité linéaire (O(n)). Avant d'expliquer chaque méthode, quelques notions sont à expliquer. Tout d'abord, nous attribuons à chaque sommet de F la valeur du plus petit arc adjacent à ce sommet. On note F-(x) la valeur ainsi attribuée au sommet x. Notion de « lowering d'arc ». Le lowering d'un arc u(x,y) correspond au remplacement du poids de u par le minimum de F-(x) et de F-(y). En d'autres termes u prendra le poids de l'arc adjacent ayant le poids

minimal. Intuitivement, il est compréhensible qu'on ne puisse pas faire de « lowering » de tous les arcs, ça ne mènerait à rien. Ainsi, pour sélectionner les arcs à « abaisser » nous allons vérifier une propriété sur chaque arc. Si cette propriété retourne « True » sur l'arc u, l'abaissement sera effectué sur u, dans le cas contraire, il ne sera pas fait.

2.2.1 Première propriété : Border-thinning

Grace à l'ajout des poids sur les sommets, on peut catégoriser les arcs en 3 sortes. Les arcs « séparant localement » sont les arcs dont le poids est supérieur aux deux sommets qu'ils relient. Les arcs « internes » sont égaux aux deux sommets qu'ils relient. C'est par exemple le cas des arcs à l'intérieur des minimums de F Les arcs de bordures sont les arcs qui sont égaux à l'un des sommets qu'ils relient, et supérieur à l'autre sommet. Puisque le poids des sommets est le minimum des arcs adjacents à ce sommet, un arc ne peut pas être inférieur au poids d'un sommet adjacent. Les 3 catégories ci-dessus couvrent donc bien l'entièreté des possibilités. La propriété de lowering est donc la suivante. Un arc devra être « abaissé » si et seulement si il appartient à la catégorie « arc de bordure ». Tant qu'il existe au moins un arc qui respecte cette propriété (c'est-à-dire, tant qu'il existe un arc appartenant à la catégorie « arc de bordure ») , le processus de lowering devra être poursuivi. Ce processus est appelé le B-Thinning (pour Border-Thinning) et lorsqu'il sera terminée, graphe obtenu, qui ne pourra plus être abaissé d'avantage, est un graphe dont l'union des minimum est appelé B-Kernel. (l'abaissement successif aura abaissé les poids d'un grand nombre d'arc de telle sorte à ce que tous les sommets fassent partie d'un B-Kernel). Le B-Kernel est tout simplement le MSF de F.

Cette méthode est intéressante du fait qu'elle est executée en local (sur chaque arc, sans se soucier du reste). Elle peut ainsi être exécutée en parallèle. Cependant, si l'exécution en parallèle n'est pas désirée, on se rend compte que la complexité de l'algorithme est est bien trop élevée (O(E2)). E (pour « edge ») étant le nombre d'arc du graphe.

2.2.2 Deuxième propriété : Minimum-thinning

Cette propriété est un cas particulier de B-Thinning. Si l'on exécute « à la main » un B-Thinning, on se rend compte très rapidement que si on vérifie la propriété (d'arc

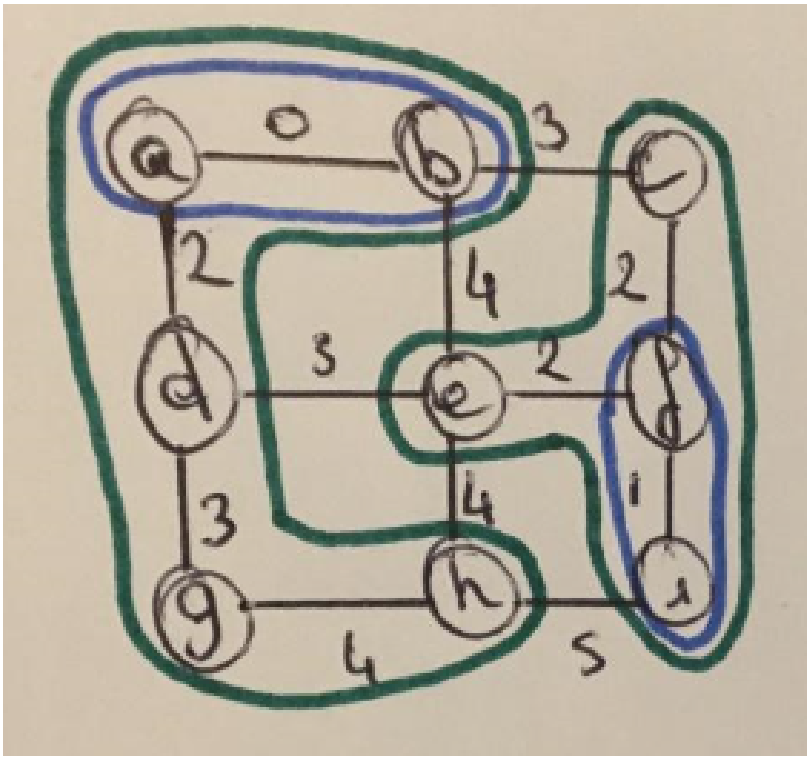


Figure 2.9: Exemple en étapes : pour chaque étape : en bleu les minimas, en vert, l'arc de bordure « abaissé ».

de bordure) seulement sur les arcs adjacents aux minimums de F, le lowering ne sera effectué qu'au plus 1 fois sur chaque arc... Par exemple, sur l'exemple précèdent (B-Thinning), on comprend rapidement que l'étape deux est de trop, qu'elle ne sert pas réellement a quelque chose, puisque l'arc est a nouveau modifié à l'étape 5... Ainsi, l'efficacité du B-thinning est augmentée est l'algorithme ne parcourra qu'une seule fois tous les arcs de F. On obtiendra donc un algorithme en complexité linéaire, pour des résultats similaires au B-Thinning. Cet amincissement s'inscrit donc parfaitement dans une optique de traitement séquentiel de l'image. En revanche l'amincissement B-Thinning est un amincissement local, et peut être utilisé en parallèle, c'est celui que nous préférerons et que nous utiliserons au cours de ce projet. Par ailleurs, un enjeu majeur du projet est de réaliser cet algorithme d'amincissement "par bloc". En effet, si nous avons une image de taille énorme, ne rentrant pas en un seul bloc dans la mémoire vive, nous devrons traiter l'images en plusieurs parties d'images (plusieurs "blocs") pour ensuite fusionner tous les morceaux. Mais n'oublions pas que nous travaillons dans un edge-weighted graphe. En effet, chaque bloc est lié avec des arcs ayant un poids, et ce poid est "mandatory" au cours

du processus de fusion des blocs. Tout d'abord, ce que nous remarquons à la fin des deux processus d'amincissement cités précédemment, c'est qu'à la fin de celui ci, il n'y aura plus aucun arc de bordure. Il ne restera plus que des arcs internes (qui constitueront les minimas) ainsi que des arcs séparants. A quoi peut nous servir cet information triviale ? Eh bien, au cours du processus de fusion des deux blocs (qui ont, je le rappelle, déjà été amincis de leur côté) nous allons abaisser uniquement les arcs de bordures appartenant à la coupure entre les deux blocs. Mais ce n'est pas tout ! Une fois que l'arc a été abaissé, il faut remettre à jour les deux bloc et ré-effectuer un B-Thinning sur les deux blocs et cela, pour être sur qu'un minimum dépassant d'un bloc soit sur un même plateau ! (C'est à dire : que chaque arc sur un même minima possède le même poid, peu importe le bloc dans lequel il se trouve).

2.3 QBT

2.3.1 Obtenir QBT

Pour obtenir une ligne de partage des eaux hiérarchique, il faut qu'on fasse la construction d'un arbre représentant l'image qui s'appelle QBT. La méthode pour produire ce graphe sera présenté ici.

QBT permet principalement quatre choses:

- Sa complexité est quasi linéaire, il est donc très rapide
- Il est possible d'obtenir la hiérarchie des zones quasi-plates à partir de cet arbre linéairement
- Il est possible de trouver les arêtes appartenant à la ligne de partage des eaux en temps linéaire
- Il est possible de calculer QBT par blocs

Tout d'abord, il est utile de réduire la dimensionnalité du problème. Nous voulons éliminer le plus d'arêtes que possible tout en conservant les informations nécessaires pour la suite.

Cela est fait en utilisant Kruskal. Mais cet algorithme présente un problème de taille, sa complexité trop importante.

Heureusement pour nous, il est possible de calculer QBT de deux manières: A l'aide de l'algorithme Union-Find, nous obtenons un arbre contenant des informations utiles, mais son calcul est relativement lent. $O(V^2)$ En utilisant la structure de données Tarjan Union-Find, l'arbre obtenu n'est pas utile. Son calcul est très rapide. $\tilde{O}(n)$

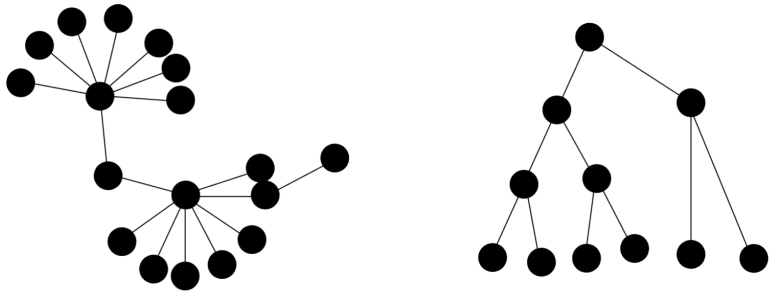


Figure 2.10: A gauche la structure Tarjan Union-find, à droite QBT. Nous pouvons voir qu'il est difficile d'extraire des informations du premier graphe

Nous voulons un algorithme rapide et donnant un arbre utile. Nous combinons donc les deux algorithmes pour ne garder que le meilleur des deux mondes, au prix d'un encombrement mémoire plus important. Deux arbres vont être créés, dont QBT, qui nous est utile.

2.3.2 Exemple de calcul de QBT

1. Des nodes sont créés pour chaque noeud du graphe si celui-ci n'est pas déjà présent dans le graphe.

2. nous ajoutons maintenant les arêtes par ordre croissant, comme dans le Kruskall normal. Ci dessous, on ajoute les arêtes A, B et F, I . D n'étant pas dans notre arbre, nous l'ajoutons. A, B étant présents dans QBT, nous faisons en sorte que A, D soit le père de A, B et D.

3. Même chose pour a, b.

4. Nous continuons ce processus d´ajout de nodes, jusqu'à ce qu'il soit impossible de rajouter d'arêtes sans créer un cycle dans le graphe. La détection de cycles se fait

grâce a l'algorithme union-find. H, I n´est pas ajouté à QBT car il est impossible de rajouter une arête sans créer de cycles, nous nous stoppons donc.

5. Nous voyons que beaucoup d'arêtes ont étés enlevées de notre arbre QBT, ce qui nous permet de réduire la dimensionnalité du problème.

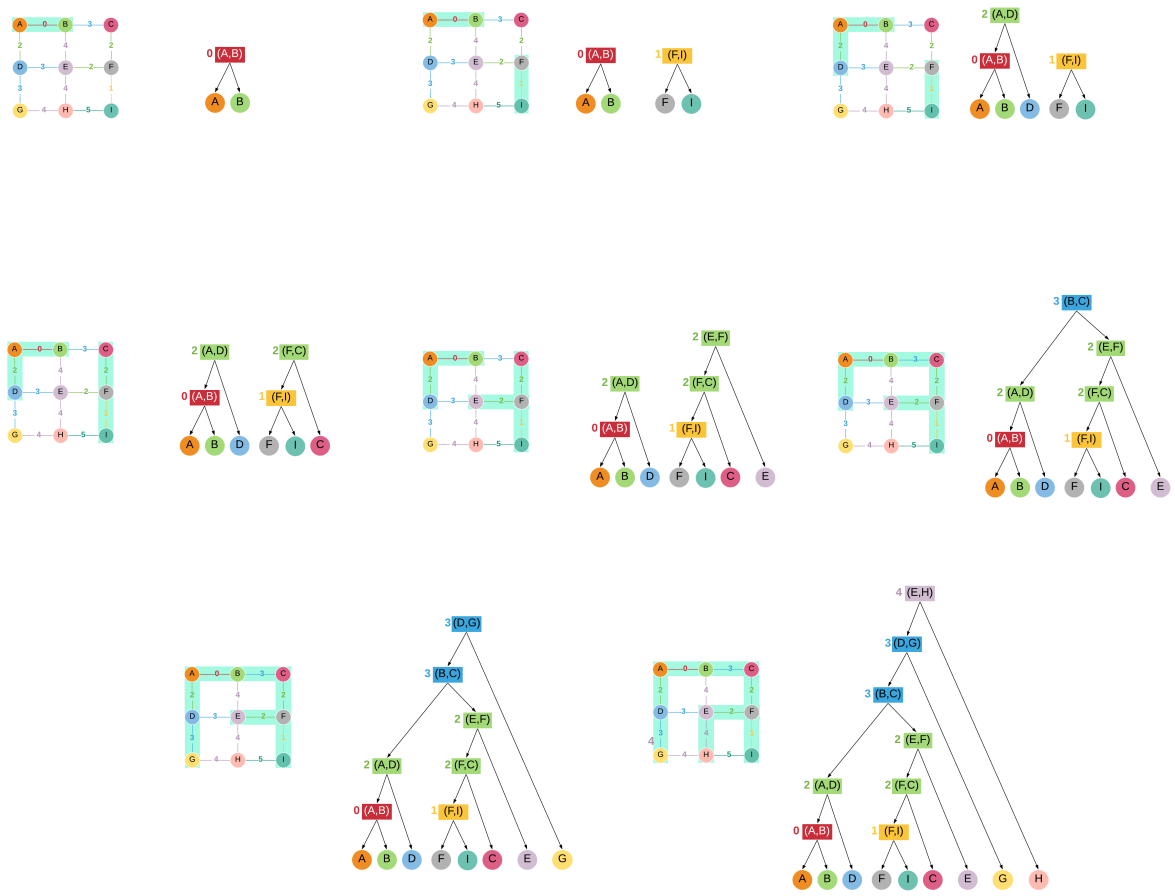


Figure 2.11: Procédure du algorithme de construction d'arbre QBT partant du Kruskal

2.3.3 Obtenir une hiérarchie des zones quasi plates

La hiérarchie des zones quasi plates nous permet d'obtenir une ligne de partage des eaux hiérarchique, mais nous n'avons pas encore implémenté la partie hiérarchique.

Pour calculer cet arbre, il faut savoir qu'à chaque noeud de QBT, un poids est assigné (0 pour les feuilles, ...). Il suffit alors de "réunir" ces noeuds de même poids.

Voici un exemple:

Partie de QBT

Après réunification des nodes de même poids: z

2.3.4 Obtenir un wateshed par flooding

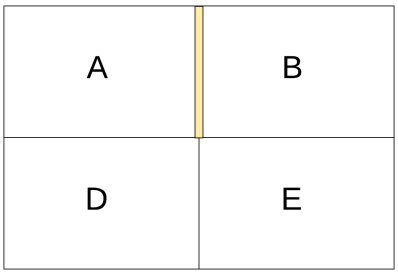
Pour obtenir ce watershed, nous partons des minimas "tout en bas de notre arbre", puis nous "augmentons le niveau de l'eau" progressivement. Ici, les groupes C,D et G,H sont deux bassins différents. Ces bassins se rejoignent sur une arête du watershed cut. Nous remontons donc le graphe de cette manière.

Ici en rouge les deux arêtes du watershed cut du graphe obtenu via QBT

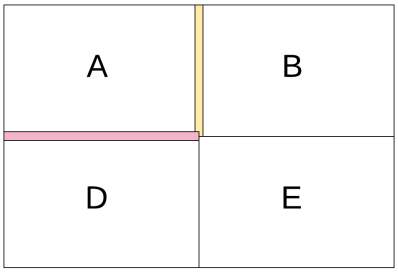
2.4 Merging of the Blocks

After producing the hierarchy of each block by using the QBT method, we are now ready to merge the blocks together.

We created a process called "Server" which manages the merging of all blocks. It will initiate the merging process by merging Block A with its neighbours on the right and on the bottom (Figure 2.20). After, we do the same for Block 1 and its neighbours and so on. We do this merging one by one until all blocks are merged together.



(a) Merging block A with its neighbour on the right (block B)



(b) Merging block A with its neighbour on the bottom (block D)

Figure 2.12: Generation of the graph

The merging of two blocks happens on their touching border. We select two nodes, x and y, one of each block, that are connected by the border we are now processing. For each node, we generate their relative border tree (BT) and send them both of these to the merging process.

To examplify this procedure, we are going to use blocks A and B and their connecting edges. Figure 2.13 shows us that blocks A and be can be merged with block B in three different edges: C-J, F-M, I-P. In order to follow the hierarchy using Kruskal, we sort the edges and select the one with the least value first. Therefore, we are going to start the merging with the edge F-M (Figure 2.14).

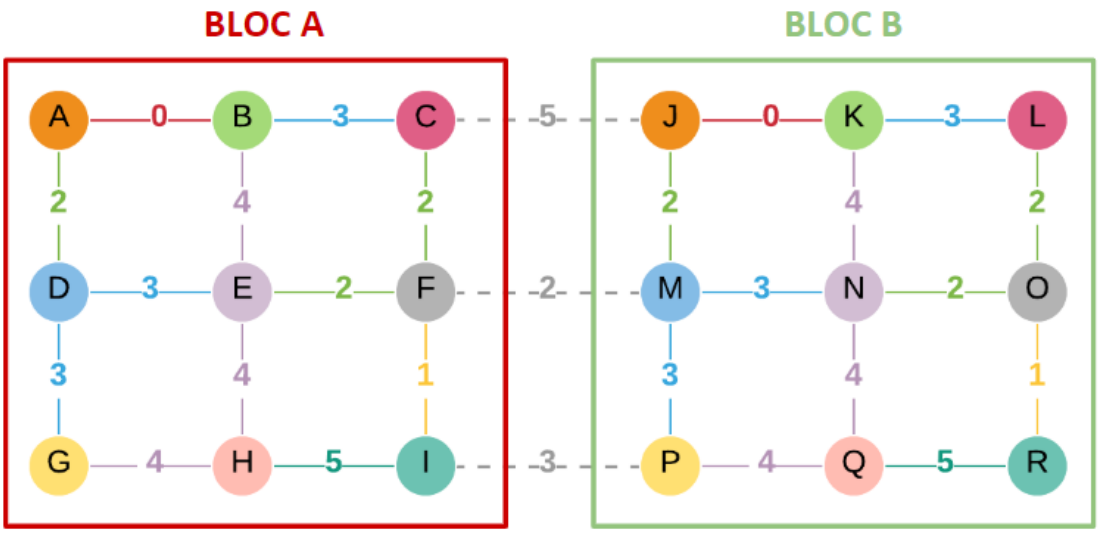


Figure 2.13: Touching border of blocks A and B

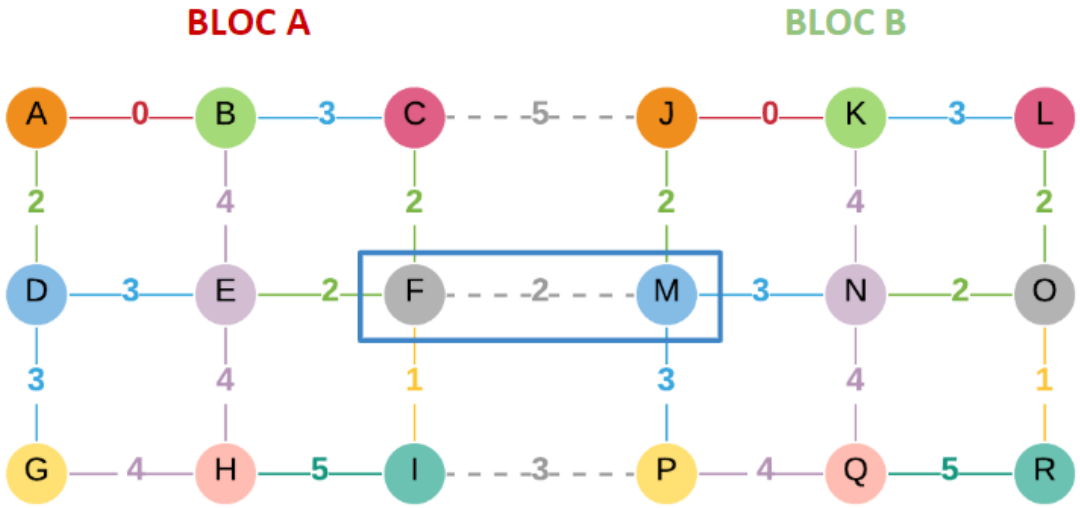


Figure 2.14: Choosing edge F-M

2.4.1 Calculation of the Border Tree (BT)

The BT is the sub-tree of the hierarchy of the block which includes only the leaf we are searching, in this case, F or M, and all the parents until we reach the root. We can see

in Figure 2.15 the construction of the boundary tree of the both of the nodes in their respectively QBTs.

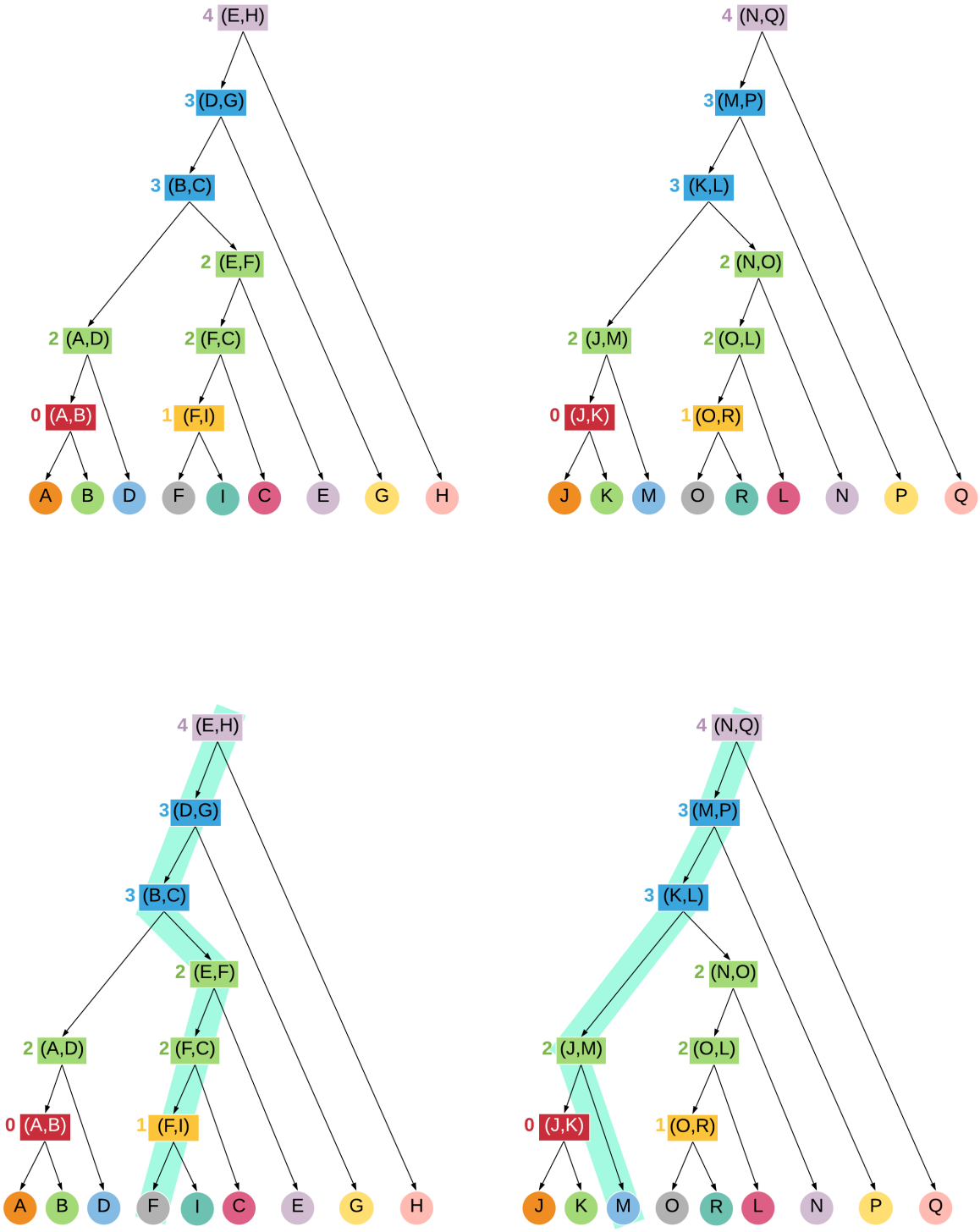


Figure 2.15: Calculating the BTs of F and M

We obtain the BT [4] of each node and the Server has the information of the altitude of the merging (in this case 2). On Figure 2.16, we have the boundary trees of F and M, and the values on the side of each node represents its altitude.

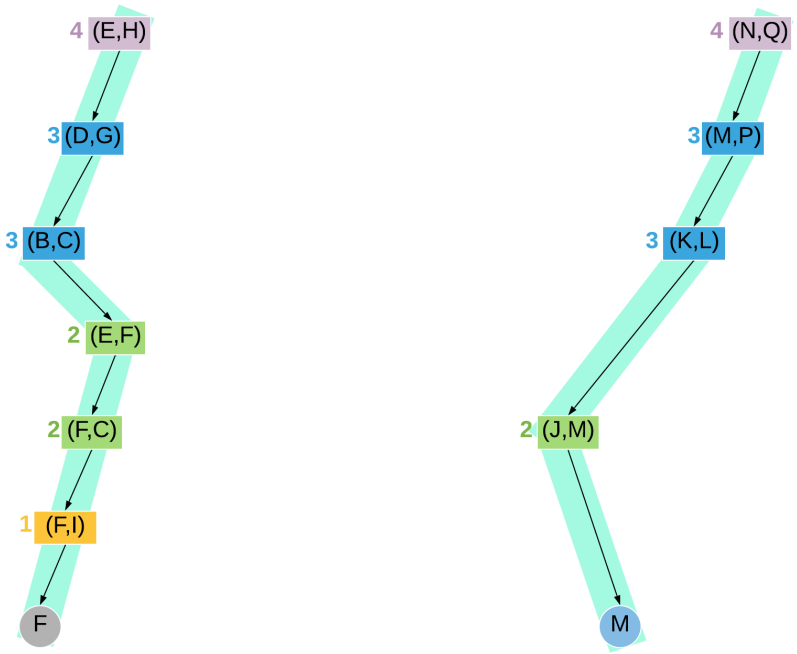


Figure 2.16: Boundary trees of F and M

2.4.2 Choosing the position of the new node

The first part of the merging is to find where the new node is going to fit. When we reach an altitude which is higher than the altitude of the merging, we select its children to be the children of our new node. The new node is called (F,M) and has altitude equal to 2 (Figure 2.17).

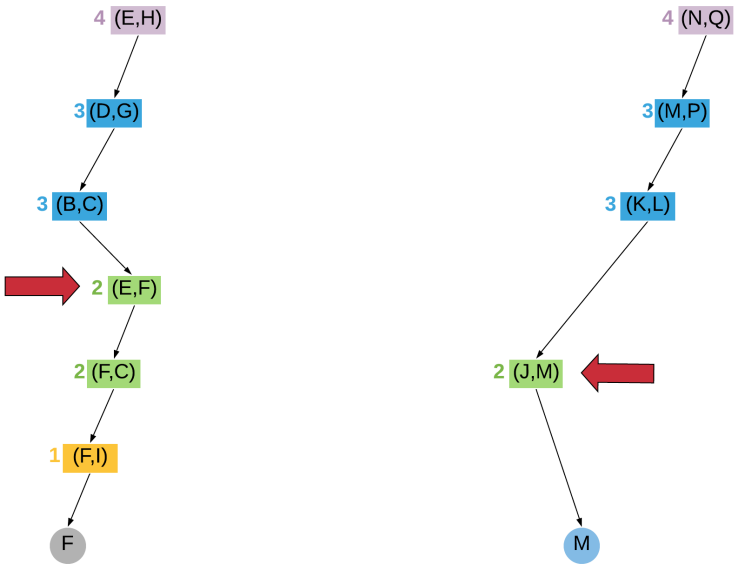


Figure 2.17: Merging F and M - Finding the children of the new node

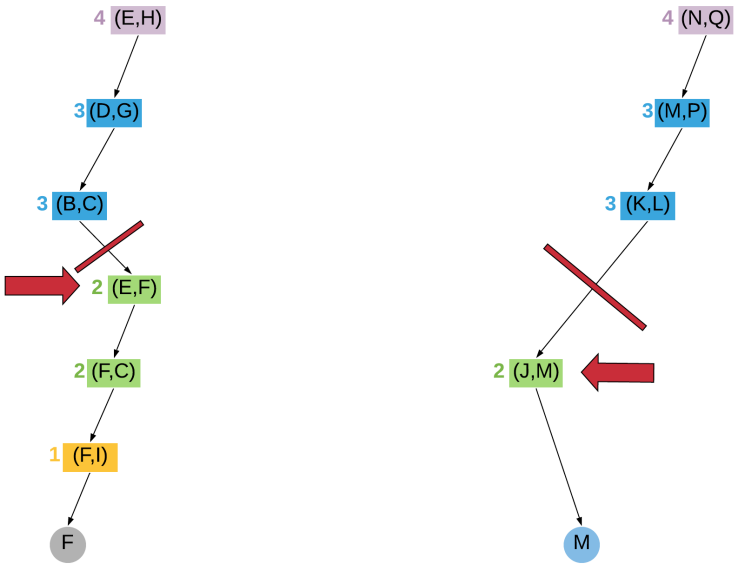


Figure 2.18: Merging F and M - Unbinding the the parents of altitude higher then 2

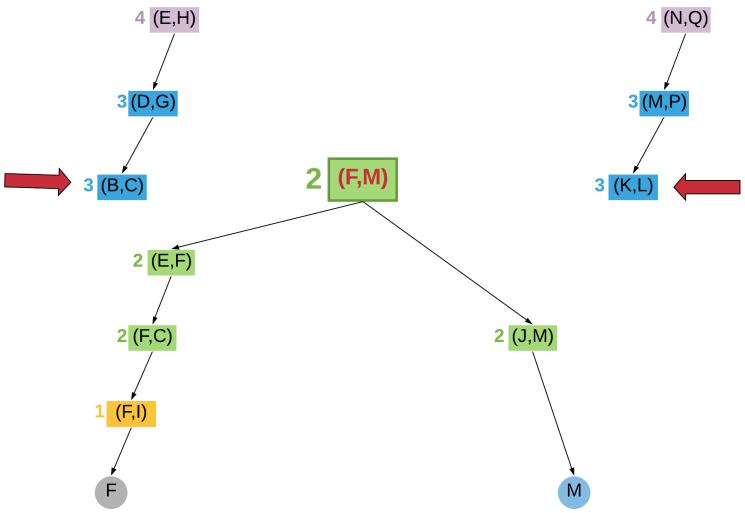
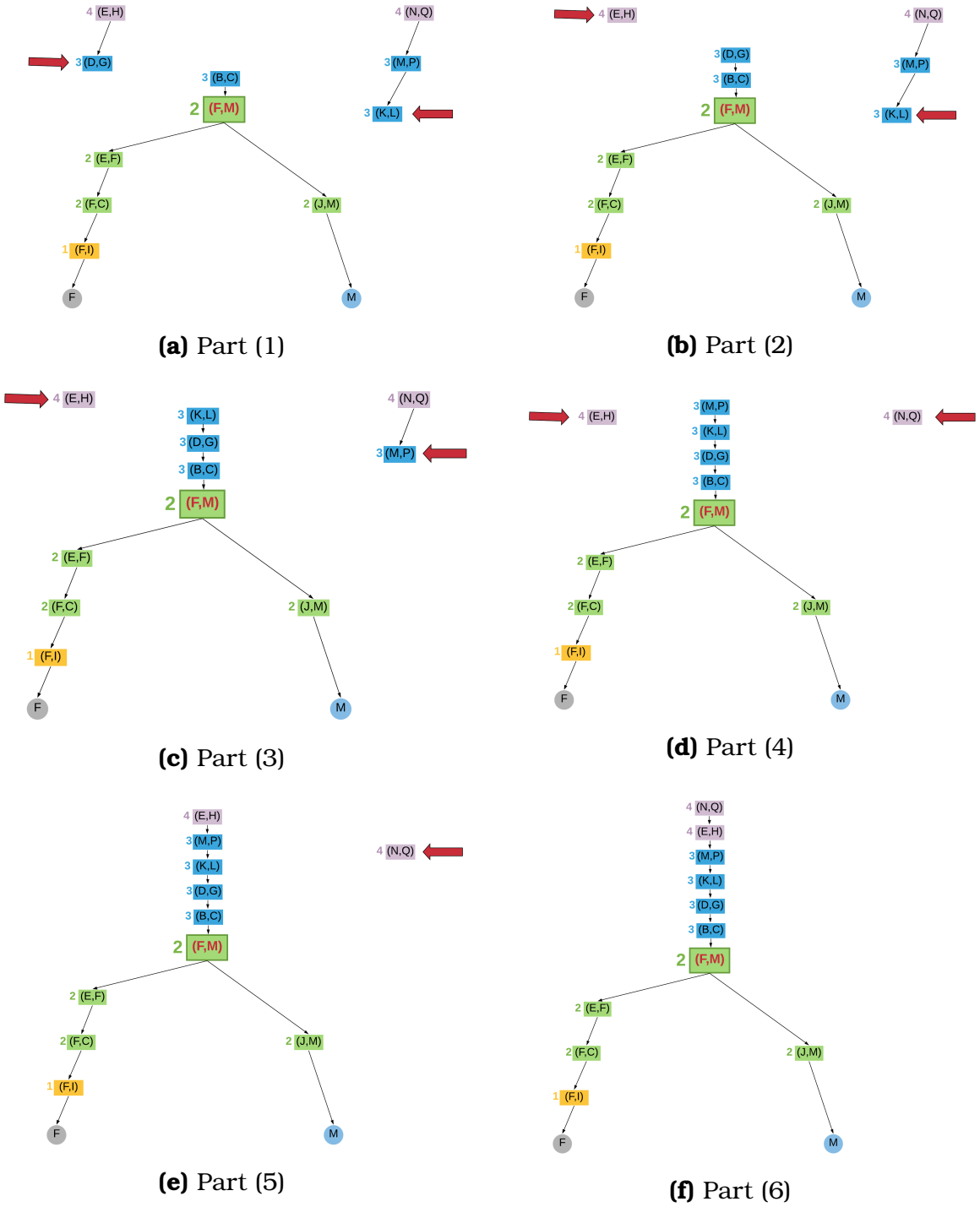


Figure 2.19: Merging F and M - Merging the children with the new node

After that, we remain with the other nodes which are now going to be selected be the future parent of the new node (F, M). The selection happens in order of preference. First, we compare the altitude, and the lowest one wins. If the altitudes are the same, we select in alphabetical order. In Figure 2.19, we can see how the whole process happens.



(a) Part (1)

(b) Part (2)

(c) Part (3)

(d) Part (4)

(e) Part (5)

(f) Part (6)

Figure 2.20: Merging F and M - Merging the remaining nodes

With this result, we now generate a new boundary tree for each of the starting nodes, but now they are related to the previous result tree MT. We can see both of the boundary

trees in Figure 2.22. Now, we have to send this information back to the original blocks in order to update their trees.

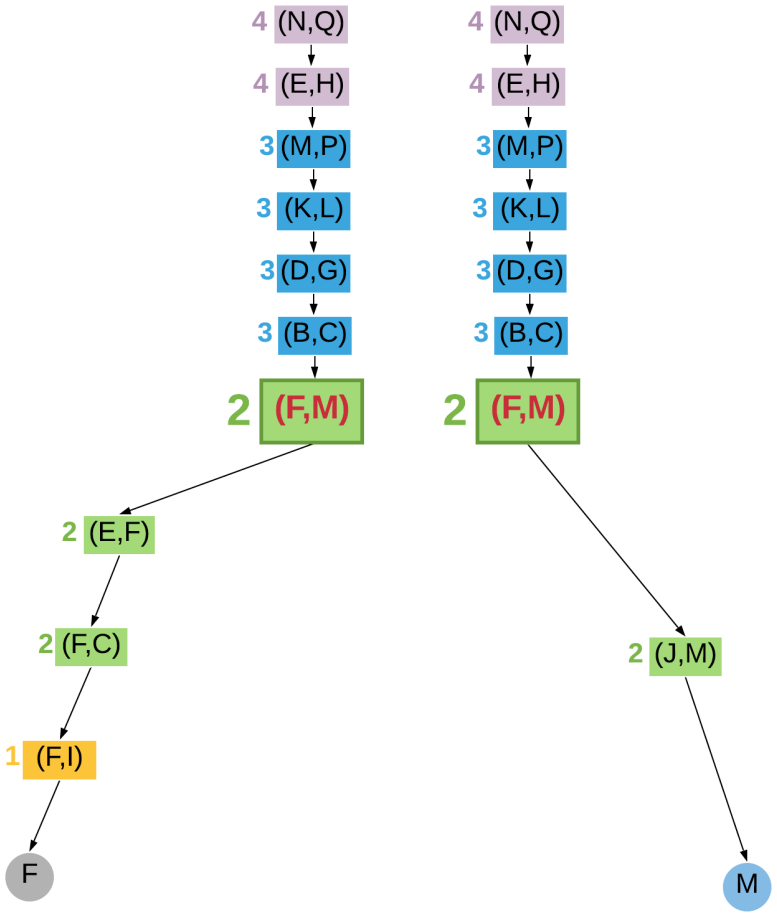


Figure 2.21: Merging F and M - Updated boundary trees

We repeat this process for each edge in the border until we have all of the border covered. In the end, each block will have in its tree all the information that it needs to calculate the watershed.

2.4.3 Updating the blocks

At the end of these two steps, we have to return the updated boundary trees to their respective blocks. We will compare the old sub-tree with the new one, in order to detect the new nodes that are now present.

Every time we find a new node that wasn't on our QBT previously, we are going to add them. The final result for block A can be seen in Figure **??**.

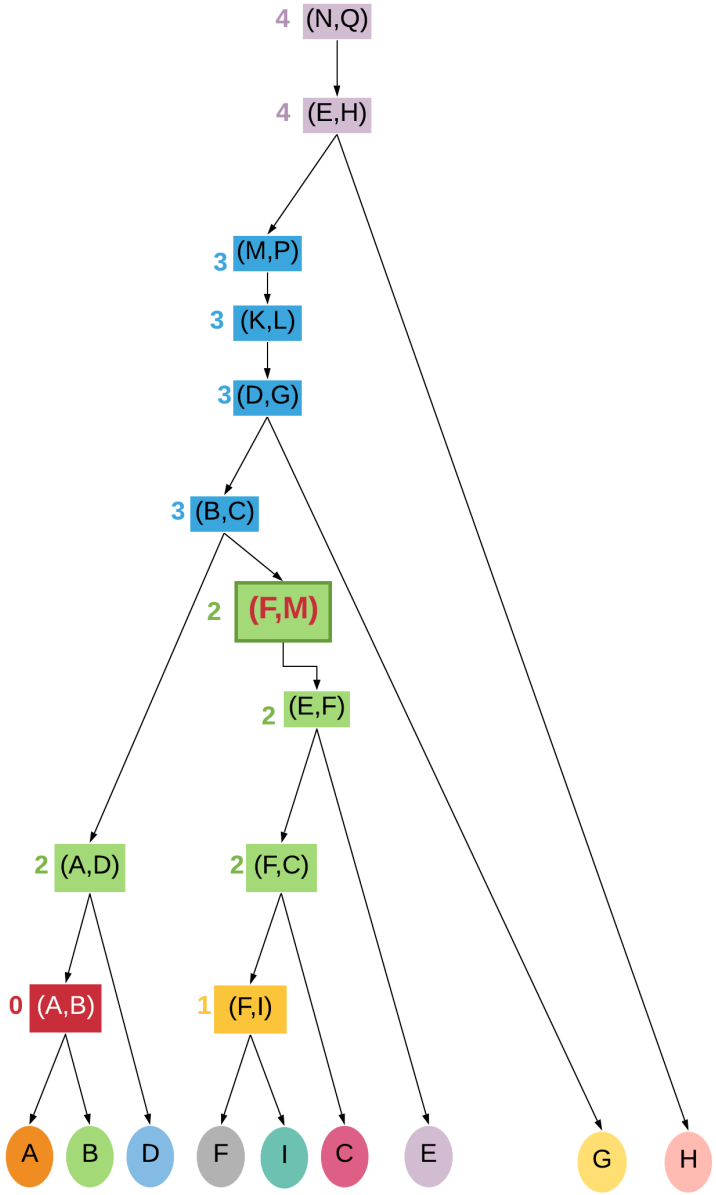


Figure 2.22: Merging F and M - Updated QBT block A

There are some errors that can occur due to the merge. But, as the updating the block procedure doesn't take any nodes out of the tree, we need to have a function that maintains the respect to the rules of the Kruskal's algorithm, and so to the creation of the QBT as well.

2.4.4 Refactoring the QBT

The idea of the production of the QBT is to have a hierarchy based on order of the edges that are selected by Kruskal's algorithm for calculating the minimum spanning tree. But, when we produce the merging of the blocks, there can be some cycles created by our new merged tree that can not be detected while merging.

Therefore, after each complete border is merged, we get the sub-trees of each block of the leafs we have used in the merging process. We look through all the nodes searching for nodes that are parents of only one child, and we add them to their respective list called w_i and w_j, where i and j are the indexes of the blocks we are merging now.

Then, we search for the nodes that have the same child in both trees. This node will be then deleted from each block, because it is not merging any children, but only making a cycle.

2.5 Calcul d'attributs

Les attributs sont des caractéristiques qui sont attribués à chaque noeuds du QBT. Ceux-ci sont définis à partir de caractéristiques de la surface topographique, obtenue à partir de l'image de base. Ils servent de repères pour la réalisation d'une ligne de partage des eaux.

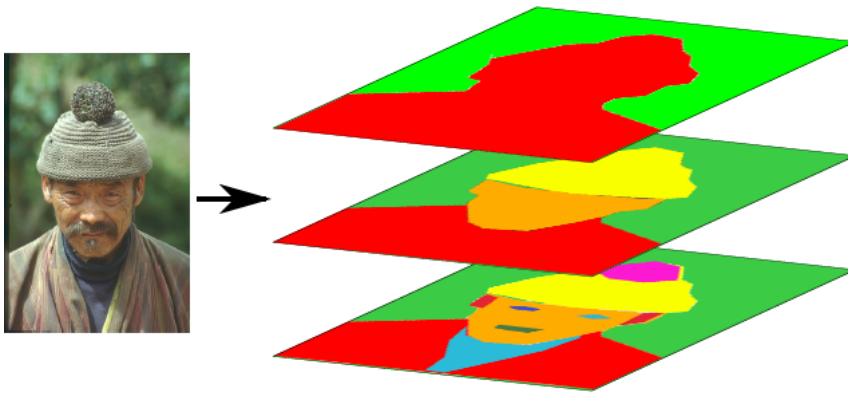


Figure 2.23: Sur cet exemple on peut voir l'évolution de la ligne de partage.

2.5.1 Calcul de la Surface

2.5.1.1 Pour un QBT

La surface pour un noeud est donc le nombre de pixel appartenant au bassin du noeud. Pour pouvoir le calculer il faut donc les valeur des deux enfants du noeud. Donc pour un arbre vide il faut parcourir le QBT des feuilles de l'arbre (représentant les pixels) jusqu'au noeud et calculer pour chaque noeud la surface.

Dans notre cas, les feuilles de l'arbre on un attribut surface de base à 1. Les noeuds qui suivent sont alors calculé à partir l'addition de la surface des deux enfants.

2.5.1.2 La mise à jour des surfaces lors de la fusion

Avant cette étape, on a déjà effectué une première fois le calcul de la surface sur les deux QBT qui s'apprêtent à fusionner. Il y a donc QBT arbres de bordure qui sont créé à partir des deux QBT pour fusionner à travers la création d'un nouveau noeud.

Jusqu'à ce nouveau noeud il n'y a pas de changement et ensuite comme pour un QBT classique on calcule sa surface. On défini deux delta un pour le premier et un pour le second bloc qui est la différence entre la surface du nouveau noeud et la surface du noeud précédant qui provient soit du premier soit du second bloc.

Pour les noeuds suivant sur l'arbre de fusion, on additionne donc leur ancienne surface avec le delta correspondant à leur bloc respectif.

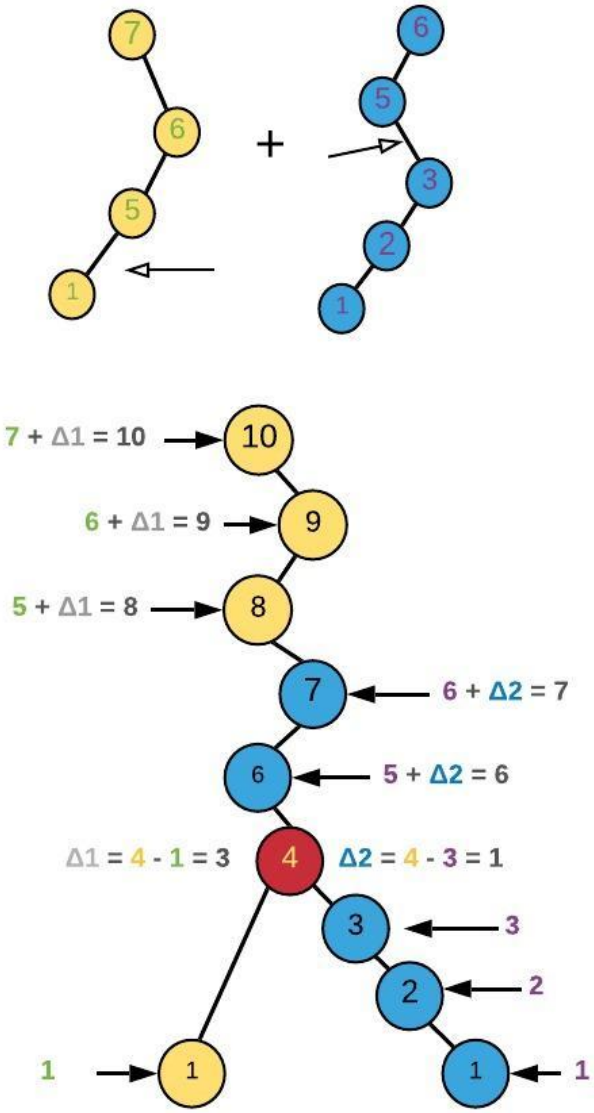


Figure 2.24: Voici un exemple de mise à jour de surface d'un graphe au moment de la fusion.

2.5.2 Calcul de la hauteur

2.5.2.1 Pour un QBT

La hauteur d'un noeud dépend de son positionnement dans l'arbre. Nous commençons par attribuer récursivement à chaque sommet de chaque QBT une hauteur. Cette hauteur est égale à 1 pour les feuilles et pour les autres sommets elle est égale à la hauteur maximale

de ses enfants, que l'on incrémente de 1.

2.5.2.2 La mise à jour des hauteurs lors de la fusion

Lorsque nous créons un arbre de fusion nous mettons à jour les noeuds de cet arbre en suivant la même règle récursive: La hauteur du noeud est égale à la hauteur maximale entre son ancienne hauteur et le calcul de la hauteur pour le noeud dans le QBT de fusion. En ce qui concerne les feuilles de l'arbre elle sont aussi initialisé à 1 comme pour un QBT classique.

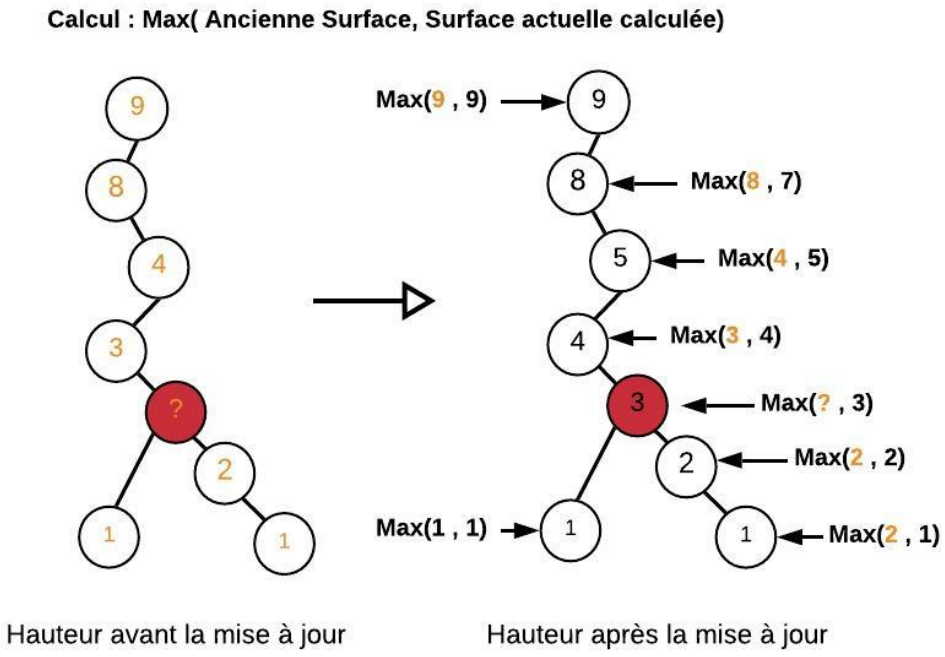


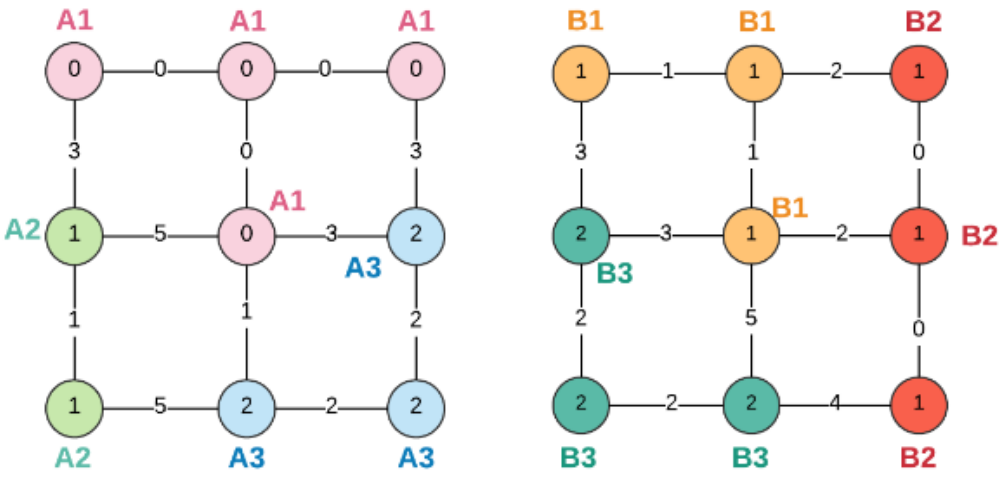
Figure 2.25: Voici un exemple de mise à jour de hauteur d'un graphe après la fusion.

2.6 Etiquetage des composantes connexes

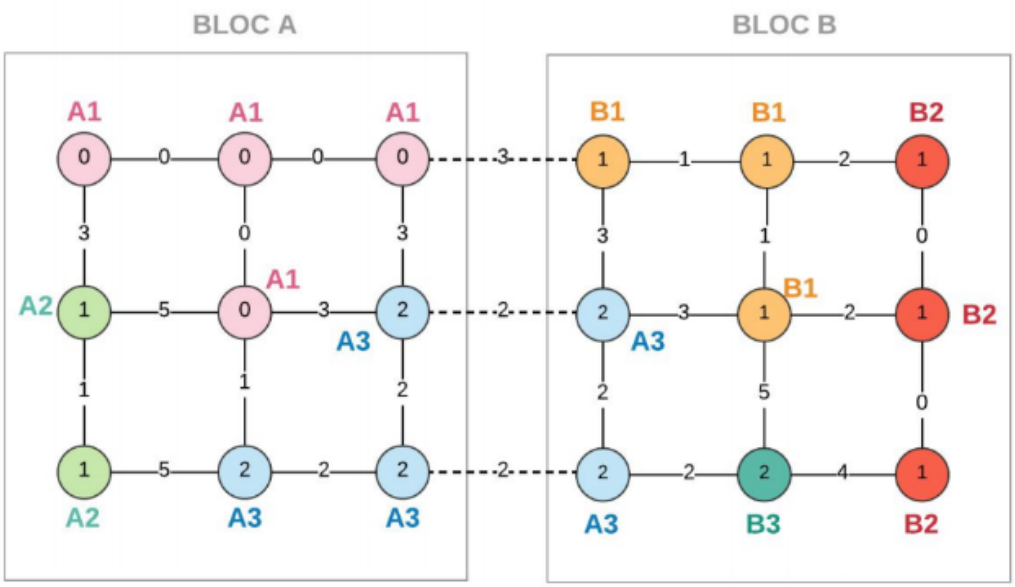
Une composante connexe d'un graphe est un sous-graphe dont, pour tout sommet appartenant à cet ensemble, il existe un chemin permettant d'aller à tout autre noeud de ce sous-graphe.

Pourquoi faire un étiquetage en composante connexe ?

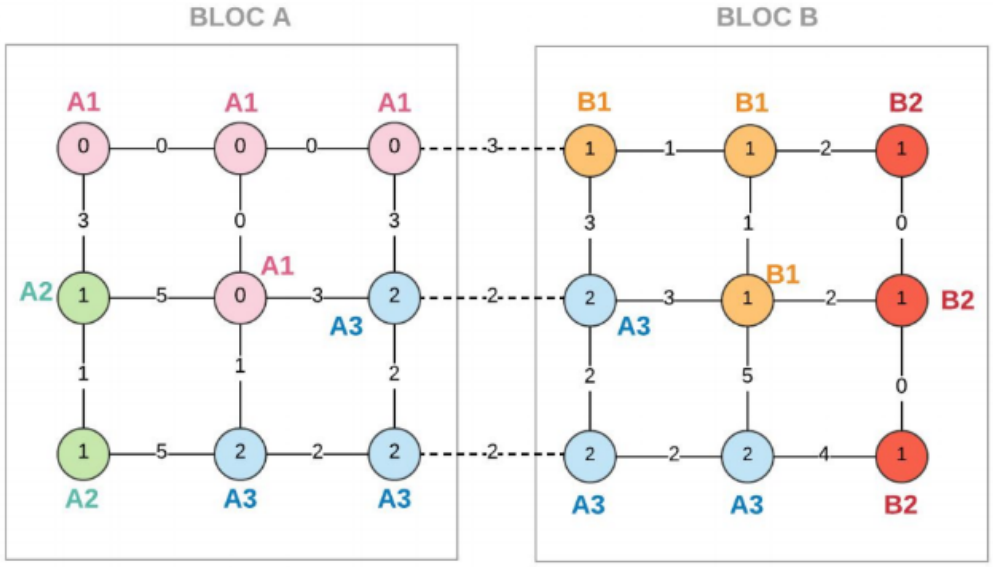
Soit 2 blocs pour lesquels on a calculé indépendamment l'un de l'autre les composantes connexes. Dans le cas du B-Thinning, les composantes connexes sont les minimas étendus. Les labels sont uniques.



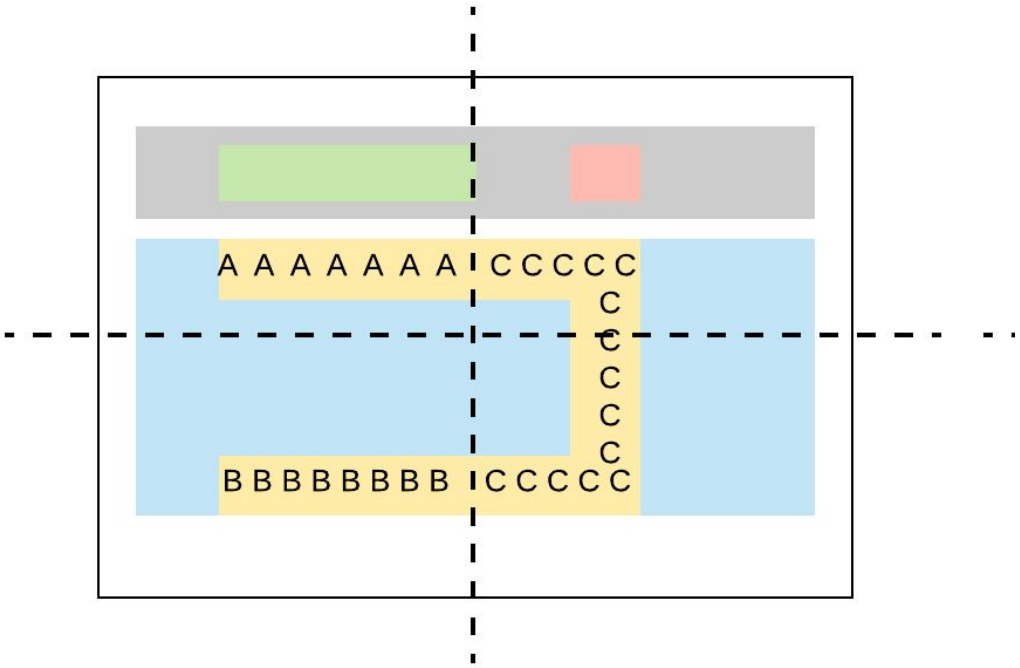
On cherche à merger les deux blocs. On merge les labels des arcs internes de la coupure. Si l'arc est un arc interne, alors le label de l'un des deux noeuds adjacent à l'arc prend la valeur de l'autre noeud adjacent.



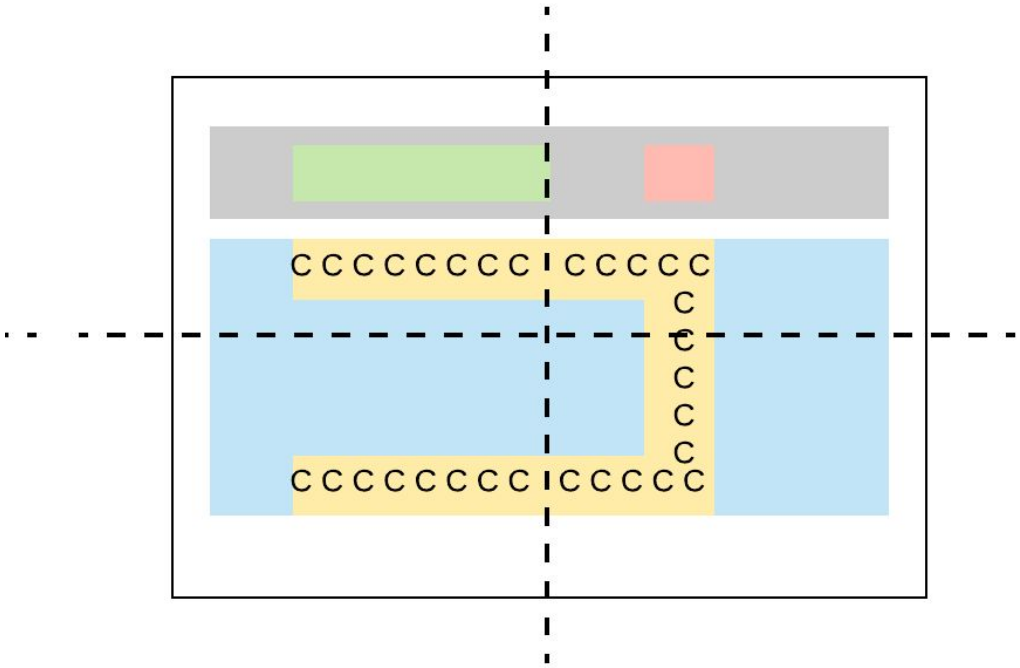
On mémorise les changements dans une table, puis on parcours le bloc du label qui a été modifié. On actualise les labels avec les informations contenues dans la table. Dans notre cas, on a retenu dans la table qu'il fallait modifier tous les B3 en A3 dans le bloc B



Cette technique, bien que peu efficace en terme de temps de calcul car on ré-actualise la totalité des labels, fonctionne toujours. Par exemple, dans l'exemple ci-dessous, le merging du bloc supérieur gauche et inférieur droit aurait montré que les labels A et C sont différents.



Cependant, on constate, en les fusionnant avec les deux autres parties, que les labels A, B et C forment un seul et même label et donc une seule et même composante connexe :



Comme expliqué précédemment, cette technique est très coûteuse en terme de temps de calcul : dans un des pires cas, on devrait parcourir plusieurs fois chaque blocs avant de trouver une solution. Nous avons donc cherché une piste d'amélioration sur ce point, et avons décidé d'implémenter la structure de donnée Union-Find pour classer les labels.

Cet structure fonctionne avec des ensembles, et offre deux opérations: Union, qui permet de fusionner deux ensembles; Find, qui permet de trouver l'élément représentant l'ensemble.

Pour notre problème, nous pouvons appliquer cette structure aux labels. On regarde les arcs de coupure entre les blocs, et lorsque l'on trouve un arc interne, on fait l'union des deux labels des noeuds reliés par l'arc. Lorsque tous les arcs de bordure ont été traité, on peut remplacer les labels en utilisant Find.

Ces opérations sont très peu coûteuses en terme de temps de calcul, on ne fait qu'un seul passage sur chaque arc de bordure, et un seul passage sur chaque noeud du graphe ensuite.

Chapter 3

Experiments and Results

3.1 Update of the QBT tree

For the first experiment, we calculated the time taken by the algorithm to calculate all QBTs of each block and the time taken for merging them all. Unfortunatly, the algorithm did not run as quick as expected. As we wanted to see how the method would work conserning the time, we chose to execute the experiments with an image of size 32x32 and another one of 64x64.

In the Figures **??** and **??**, we can see the results of the total time taken.

In general, we can see that the time taken to merge the blocks together, will always increase the overall time. When we have only one block is faster then if we have more.

The interesting thing is that when increasing the number of blocks we can decrease the time taken. Our intuition is that as the number of blocks rises, we are going to have trees that are smaller in height, and by doing so, we decrease the time we spend searching for nodes on the tree.

As the compute time, which is the time spent on the merging of the boundary trees, is almost constant, we can infer that what makes the algorithm slow is the functions related to updating the blocks (Suppression des node and Mise à jour de QBT), which are the ones that make a lot of operations related to going through all the tree many times.

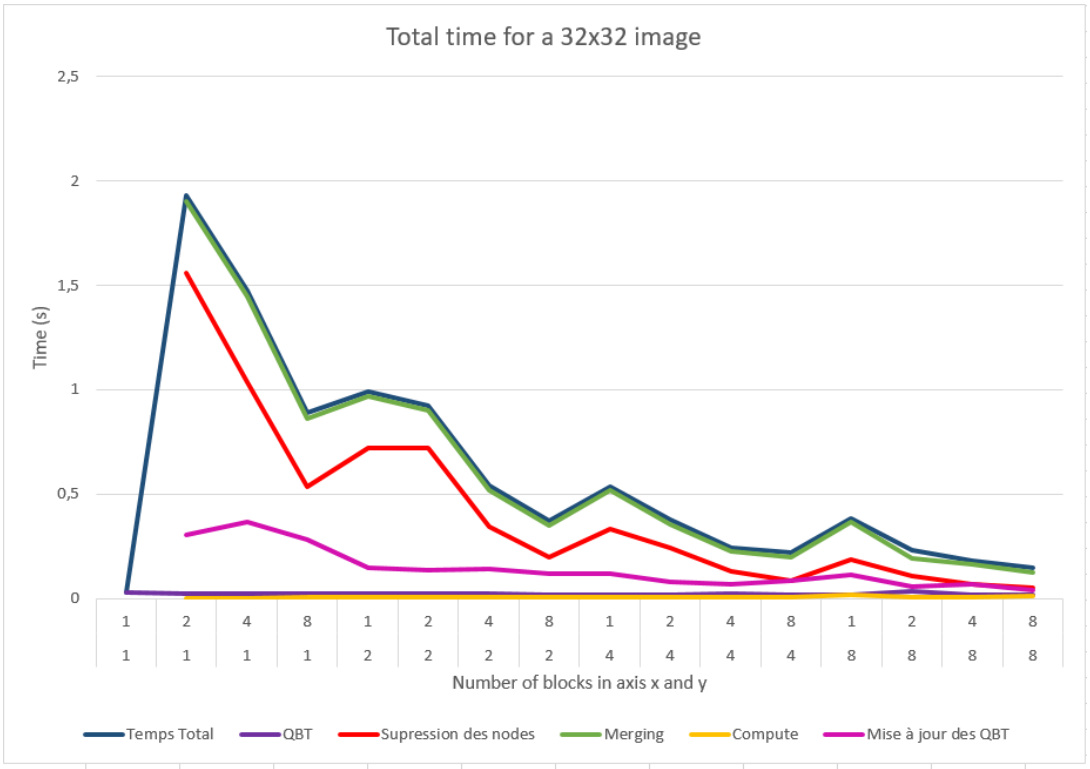


Figure 3.1: Time needed for the execution of the algorithm in an image 32x32

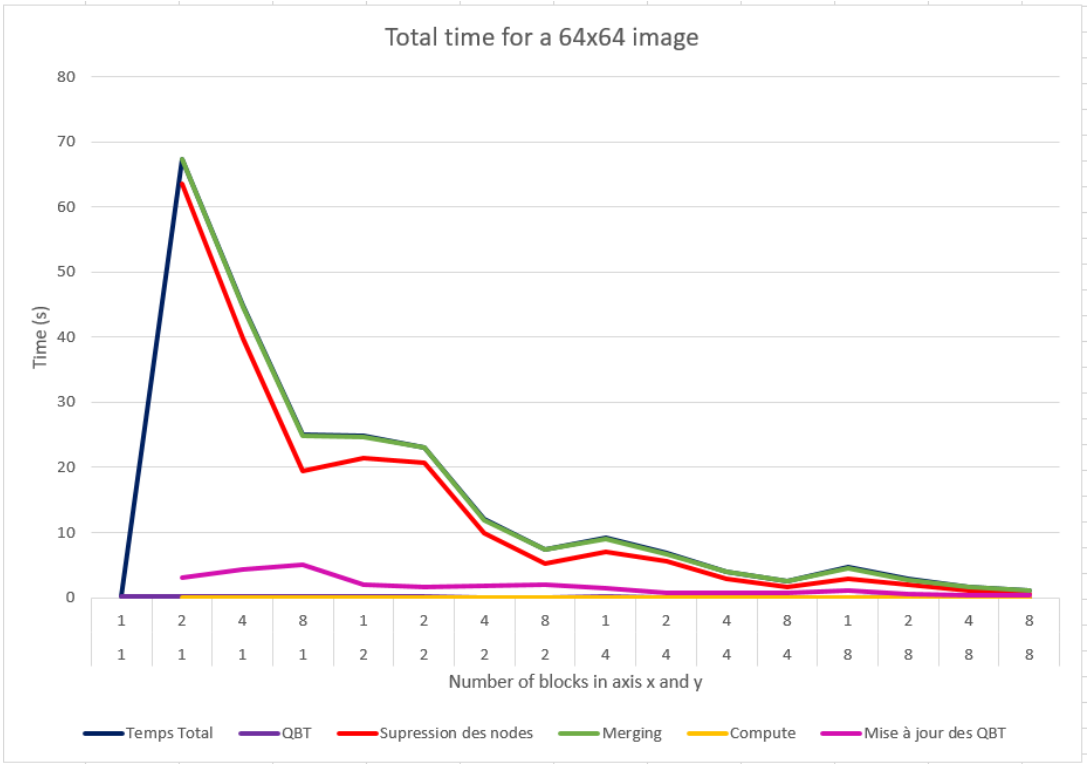


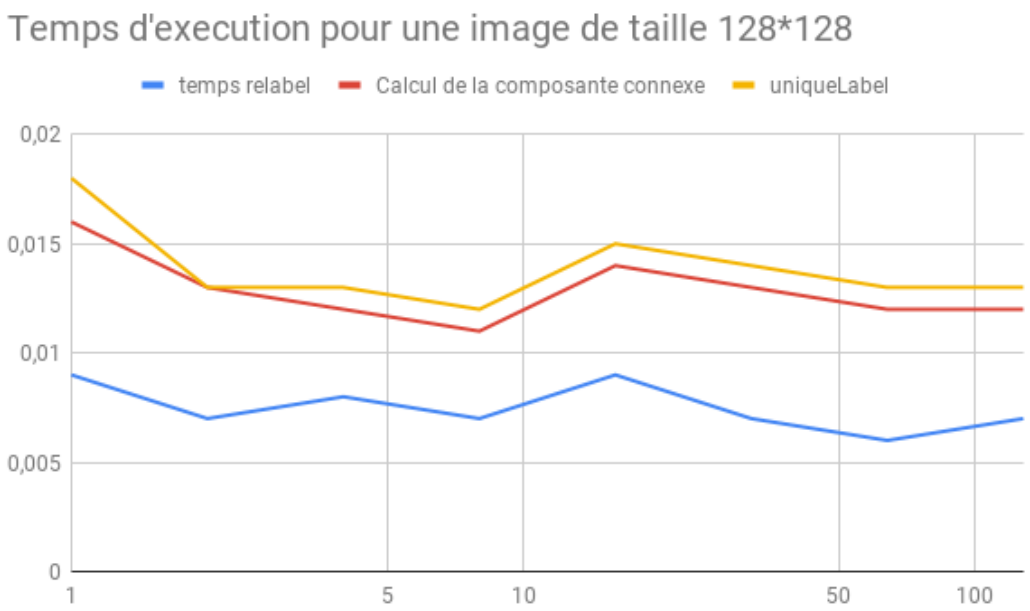
Figure 3.2: Time needed for the execution of the algorithm in an image 64x64

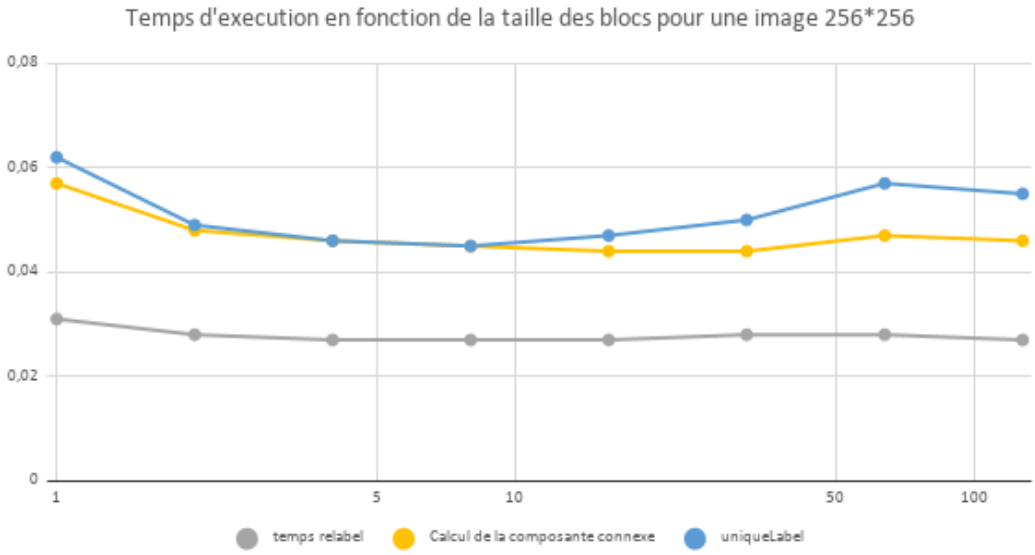
3.2 Étiquetage en composante connexe

Nous avons mesuré le temps de calcul pour l'étiquetage en composante connexe. Dans l'exécution, nous sommes limité par notre implémentation en python qui ne nous permet pas d'exécuter le code sur des images trop grandes.

Voici les résultats en en fonction de la taille des blocs.

Sur les graphiques, il y a trois courbes : Calcul de la composante connexe, qui est le temps total pris pour calculer la composante connexe pour la totalité des blocs, uniqueLabel; qui est le temps pour initialiser les labels et les renommer de manière unique à chaque blocs, reLabel qui est le temps de traitement final pour repasser sur le graphe et remplacer tous les labels.





On remarque que la taille des blocs n'influence pas dramatiquement le temps de calcul. Ce qui est en revanche intéressant de noter, c'est que le temps de calcul pour le calcul de la composante connexe par bloc peut être parallélisé, et comme on remarque que ce temps de calcul par rapport au reste est non négligeable, cela peut avoir un intérêt.

Chapter 4

Conclusion

Pour conclure, on peut dire que ce projet nous a permis d'acquérir des connaissances en traitement d'image et en théorie des graphes. Par ailleurs c'est un projet particulièrement intéressant, parce que l'algorithme que nous produisons est innovant.

4.1 Management

D'un point de vue organisationnel, nous avons divisé les tâches par binôme au premier semestre : Jules et Stela sur la partie "Hiérarchisation et Fusion" ; Cécile et Edouard sur la partie "Ligne de partage des eaux par blocs".

Au second semestre, Alexandre, Erwan et Florian ont rejoint le projet et Edouard est parti à l'étranger. L'organisation a été la suivante : - Jules et Stela (équipe 1) ont continué sur la partie Hiérarchisation et Fusion ; - Florian et Alexandre (équipe 2) ont travaillé sur le calcul des attributs ; - Cécile et Erwan (équipe 3) ont travaillé sur l'étiquetage en composantes connexes.

Jules et Stela ont du travailler en collaboration avec Florian et Alexandre puisqu'ils devaient construire leur travail sur celui déjà réalisé. Pour la partie Union Find, l'équipe 3 a travaillé avec l'équipe 1 qui connaissait déjà le fonctionnement de l'algorithme.

Nous avons rencontré notre tuteur tous les lundis après-midi pour faire le point sur nos avancements respectifs. Nous nous retrouvions également sur réaliser en équipe mob

programming).

Nous avons utilisé plusieurs outils dans le cadre de ce projet : - Slack : nous y avons créé des chaînes de discussion par thématique (hiérarchie_contours, lpe_élémentaire, lpe_hiérarchique, général, liens_utiles) pour organiser les discussions. - Github : Planning de préparations, faire du versionning de code. Nous avions également mis en place un script de qualité permettant de détecter les erreurs dans le code via des tests unitaires. - Drive : Nous y déposons les ressources utiles, le rapport, etc.

4.2 État final du projet

À la fin du projet, on a produit des algorithmes pour faire:

1. le calcul des arbres QBT pour chaque bloc
2. le calcul des attributs: surface, hauteur
3. la fusion des blocs et la mise à jour des attributs en parallèle
4. étiquetage de composantes connexes pour plusieurs blocs
5. segmenter une image par amincissement itéré (M et B Thinning) pour deux blocs.

Références

[1] J. Cousty, L. Najman, Y. Kenmochi, S. Guimarães, Hierarchical segmentations with graphs: quasi-flatzones, minimum spanning trees, and saliency maps, Journal of Mathematical Imaging and Vision, Springer Verlag, 2017.

[2] J. Cousty, G. Bertrand, L. Najman, M. Couprie, Watershed Cuts: Minimum Spanning Forests and the Drop of Water Principle, IEEE Transactions on Pattern Analysis and Machine Intelligence, Institute of Electrical and Electronics Engineers, 2009.

[3]L. Najman, J. Cousty, B. Perret, Playing with Kruskal: algorithms for morphological trees in edge-weighted graphs, C.L. Luengo Hendriks, G. Borgefors, R. Strand. International Symposium on Mathematical Morphology, May 2013, Uppsala, Sweden.

[4] Kazemier, Jan J., Georgios K. Ouzounis, and Michael HF Wilkinson. "Connected morphological attribute filters on distributed memory parallel machines." International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing. Springer, Cham, 2017.