HAL open science

Etude du passage à l'échelle des algorithmes de segmentation et de classification en télédétection pour le traitement de volumes massifs de données

Pierre Lassalle

▶ **To cite this version:**

Pierre Lassalle. Etude du passage à l'échelle des algorithmes de segmentation et de classification en télédétection pour le traitement de volumes massifs de données. Sciences de la Terre. Université Paul Sabatier - Toulouse III, 2015. Français. NNT : 2015TOU30261 . tel-01388003

HAL Id: tel-01388003

https://theses.hal.science/tel-01388003v1

Submitted on 2 Nov 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Université Toulouse 3 Paul Sabatier (UT3 Paul Sabatier)*

Présentée et soutenue le *06/11/2015* **par :**

Pierre LASSALLE

Étude du passage à l'échelle des algorithmes de segmentation et de classification en télédétection pour le traitement de volumes massifs de données.

JURY

Jordi INGLADA	Chercheur	Directeur de thèse
Jean-Philippe GASTELLU	Professeur	Président du Jury
Grégoire MERCIER	Professeur	Rapporteur
Sébastien LEFEVRE	Professeur	Rapporteur
Julien MICHEL	Ingénieur	Examinateur
Pierre GANÇARSKI	Professeur	Examinateur

École doctorale et spécialité :
SDU2E : Océan, Atmosphère et Surfaces Continentales
Unité de Recherche :
Centre d'Études Spatiales de la BIOsphère (UMR 5126)
Directeur de Thèse :
Jordi INGLADA
Rapporteurs :
Grégoire MERCIER et *Sébastien LEFÈVRE*

Sommaire

Sommaire ii
Liste des figures vii
Liste des tables ix

Introduction aux algorithmes échelonnables 8

1 Algorithmes échelonnables 9
1.1 Définition et notations 10
1.1.1 Définition d'un algorithme 10
1.1.2 Passage à l'échelle 11
1.1.3 Algorithmes et programmation informatique 11
1.2 Modélisation d'un ordinateur 12
1.2.1 Description générale d'un ordinateur 12
1.2.2 Différents types de mémoire 13
1.2.3 Modélisation d'un ordinateur dans le cadre de nos travaux 16
1.3 Complexité algorithmique 16
1.3.1 Notation de Landau 18
1.4 Structures de données 20
1.5 Techniques algorithmiques pour le passage à l'échelle 26
1.5.1 Algorithme de flots de données 27
1.5.2 Diviser pour mieux régner 27
1.5.3 Réduction de la dimension des données 29
1.6 Stabilité 31

Segmentation 32

2 Segmentation par fusion de régions 33
2.1 Introduction 33

2.2 Segmentation basée sur les régions 35
2.3 Différents critères locaux d'homogénéité 36
2.4 Différentes heuristiques pour la fusion des segments 39
2.5 Motivations pour l'unification des méthodes par fusion de régions 41

3 Algorithme GRM 43
3.1 Introduction 43
3.2 Structures de données 44
3.2.1 Représentation basée sur un graphe 44
3.2.2 Représentation d'un segment 45
3.2.3 Représentation d'un lien d'adjacence 47
3.2.4 Représentation du contour d'un segment 48
3.3 Description de l'algorithme GRM et analyse de la complexité 55
3.3.1 Initialisation des segments dans l'image 55
3.3.2 Calcul des coûts de fusion 56
3.3.3 Mise à jour du contour 57
3.3.4 Mise à jour du voisinage 57
3.3.5 Suppression des segments fusionnés 57
3.3.6 Algorithme GRM 58

4 Stabilisation de l'algorithme GRM 61
4.1 Introduction 61
4.2 Impact du tuilage 63
4.3 État de l'art 66
4.4 GRM : un algorithme piloté par les données 70
4.5 Stabilité des algorithmes de segmentation 71
4.6 Zone d'influence d'un segment 73
4.7 Calcul de la marge de stabilité 75

5 Algorithme échelonnable proposé : LSGRM 79
5.1 Aperçu général de la LSGRM 79
5.2 Suppression des segments instables 82
5.3 Stockage et chargement des graphes 83
5.3.1 Stockage des graphes dans la mémoire externe 83
5.3.2 Chargement des graphes depuis la mémoire externe 84
5.4 Ajout d'une marge de stabilité à un graphe 84

5.5 Fusion de graphes de segment 87
5.5.1 Traitement des segments dupliqués 88
5.5.2 Mise à jour des arêtes 89
5.6 Example quantitatif des complexités de l'algorithme LSGRM 90

6 Validation de l'algorithme LSGRM et expérimentations 93
6.1 Étude de la stabilité du LSGRM 93
6.2 Segmentation d'une scène complète Pléiades 96

7 Bilan sur la segmentation échelonnable 100
7.1 Conclusions 100
7.2 Perspectives 101

Classification 103

8 Contexte des travaux 104
8.1 Classification en télédétection 104
8.1.1 Classification non supervisée 105
8.1.2 Classification supervisée 105
8.2 Revue des méthodes 106
8.2.1 Classification bayésienne 106
8.2.2 Les réseaux de neurones 106
8.2.3 Les séparateurs à vaste marge 107
8.2.4 Les arbres de décision 109
8.3 Problématique 111

9 Forêts aléatoires 114
9.1 Description générale de l'algorithme 114
9.1.1 Calcul de l'impureté basé sur le critère de Gini 114
9.1.2 Détermination de la coupure optimale 116
9.1.3 Partitionnement des données et distribution aux nœuds fils 116
9.1.4 Construction récursive de l'arbre de décision 118
9.1.5 Construction de la forêt aléatoire 119
9.2 Passage à l'échelle : état de l'art 120
9.2.1 La méthode SLIQ 120
9.2.2 La méthode SPRINT 122
9.2.3 La méthode RainForest 123
9.2.4 Google PLANET 124

9.2.5 Forêt aléatoire utilisant MapReduce 126
9.2.6 Discussions 126

10 Algorithme SMART 128

10.1 Aperçu général de l'algorithme SMART 128
10.1.1 Génération des listes d'attributs 128
10.1.2 Tri des listes d'attributs 129
10.1.3 Choix de la stratégie initiale 130

10.2 Description des stratégies d'exécution dans SMART 131
10.2.1 Stratégie "En Mémoire" 131
10.2.2 Stratégie "Partitionnement En Mémoire" 136
10.2.3 Stratégie "Hors Mémoire" 145

10.3 Conclusions 149

11 Validation de SMART et expérimentations associées 151

11.1 Validation de la stabilité de l'algorithme SMART 151
11.1.1 Principales difficultés 151
11.1.2 Méthodologie proposée 152

11.2 Performance de l'algorithme SMART 154

12 Bilan sur la classification échelonnable 157

12.1 Bilan 157

12.2 Perspectives 158

Conclusions 160

13 Conclusions générales 161

13.1 Segmentation 161
13.1.1 Bilan 161
13.1.2 Perspectives 162

13.2 Classification 163
13.2.1 Bilan 163
13.2.2 Perspectives 163

Bibliographie **165**

Annexes **175**

A Description des satellites Pléiades 176

B Description des données d'apprentissage 177

C Valorisation scientifique 179

D Applications et développement logiciel 180

Table des figures

1.1 Algorithme d'Euclide pour le calcul du PGCD. 10
1.2 Fonctionnement d'un disque dur. 15
1.3 Modélisation simplifiée d'un ordinateur 17
1.4 Modélisation d'une liste simplement chaînée 21
1.5 Opération d'insertion dans un tableau dynamique 23
1.6 Suppression plus efficace des valeurs dans un tableau dynamique. . 24
1.7 Algorithme External Merge 29

2.1 Étapes génériques de la segmentation par fusion de régions 42

3.1 Densité du graphe de segments au fil des itérations 46
3.2 Représentation du graphe des segments avec une liste d'adjacence 47
3.3 Représentation du rectangle englobant un segment 48
3.4 Bordure commune entre 2 segments adjacents 48
3.5 Déplacements possibles le long d'un contour 49
3.6 Représentation des contours 50
3.7 Ensemble des déplacements possibles le long d'un contour 51
3.8 Contour du segment S_{12}. 52
3.9 Génération de l'image étiquetée à partir du contour des segments . 52
3.10 Résultat de segmentation avec l'algorithme GRM 53
3.11 Réduction de la mémoire avec l'utilisation des contours 54
3.12 2 types de connectivité disponibles dans l'algorithme GRM 56
3.13 Mise à jour du voisinage lors de la fusion de S_j dans S_i 58
3.14 Évolution du temps d'exécution de l'algorithme GRM 60

4.1 Application d'un filtre gaussien en utilisant le tuilage 62
4.2 Image utilisée pour les expériences 63
4.3 Analyse visuelle de l'impact du tuilage sur les segments résultants 64
4.4 Protocole pour la comparaison des segmentations GT et TS 66
4.5 Évolution des instances de Hoover 67
4.6 Impact du tuilage sur les segments au fil des itérations 72
4.7 La propriété de couverture 73

4.8 Zone d'influence d'un segment 74
4.9 Représentation de la marge de stabilité M_n pour une tuile 75
4.10 Zone d'influence d'un segment initial 76
4.11 Marge de stabilité M_{n+1} en fonction de M_n et n 77

5.1 Diagramme d'exécution de l'algorithme LSGRM 80
5.2 Délimitation d'une tuile dans l'image. 82
5.3 Ajout de la couronne de stabilité dans le cas où $n = 1$ 85
5.4 Fusion de 2 graphes 88
5.5 Recherche des segments dupliqués. 89
5.6 Traitement d'un segment dupliqué 90

6.1 Images satellite Ikonos et Quickbird 94
6.2 Évolution de RC en fonction de la valeur incorrecte de la marge . 95
6.3 Évolution de RC en fonction de la valeur correcte de la marge . . 95
6.4 Algorithme LSGRM : répartition du temps d'exécution 98
6.5 Résultat de segmentation d'une image Pléiades. 99

8.1 Topologie d'un réseau de neurones 107
8.2 Hyperplan SVM 108
8.3 Structure d'un arbre de décision 110
8.4 Étude bibliométrique des méthodes de classification 113

9.1 Paramètres de l'algorithme forêt aléatoire. 115

10.1 Diagramme d'exécution de l'algorithme SMART 129
10.2 Hiérarchie des stratégies de l'algorithme SMART 131
10.3 Niveau d'un arbre de décision 138
10.4 Détermination des coupures optimales de tous les nœuds d'un même niveau de l'arbre avec la stratégie PEM 141
10.5 Évolution du nombre de noeuds 142
10.6 Ajout des nœuds fils aux nœuds du niveau courant 143
10.7 Représentation de la MFU 147

11.1 Procédure pour vérifier la stabilité de SMART 153
11.2 Données utilisées pour la validation de l'algorithme SMART . . . 153
11.3 Influence du nombre de données sur le temps d'exécution. 155
11.4 Influence du nombre d'attributs sur le temps d'exécution. 155
11.5 Apport de la combinaison des stratégies sur le temps d'exécution. 156

1 Données d'apprentissage Landsat 8 177

Liste des tableaux

1.1 Hiérarchie mémoire et temps de latence en nanosecondes (ns). . . 16

5.1 Temps d'exécution et mémoire utilisée par l'algorithme LSGRM . 91
5.2 Temps d'exécution de la première segmentation partielle 91
5.3 Temps d'exécution de la seconde segmentation partielle 92
5.4 Répartition du temps d'exécution lors de la segmentation finale. . 92

6.1 Temps d'exécution de l'algorithme LSGRM 97

10.1 Simulation MFU pour différentes valeurs de M. 148

1 Description des satellites Pléiades. 176
2 Description du satellite Landsat 8. 178

Remerciements

Avant de rentrer dans cette aventure algorithmique, je tiens à remercier les personnes qui m'ont soutenu durant ces trois années de thèse et qui m'ont permis de mener à terme un travail de recherche avec des contributions importantes.

Je tiens à remercier dans un premier temps mon directeur de thèse, Jordi Inglada, Ingénieur Chercheur CNES, qui m'a accompagné durant toute cette formation et a su m'orienter dans mes recherches. Nos nombreux échanges constructifs, même à 10000 km de distance, m'ont toujours été profitables et m'ont permis de garder le cap pour remplir tous les objectifs que nous nous étions fixés. Enfin, il m'a accordé toute sa confiance lorsque je voulais approfondir de nouvelles pistes de recherche et pour cela, je l'en suis très reconnaissant. En second lieu, je tiens à remercier, Julien Michel, Ingénieur Chercheur CNES, Manuel Grizonnet, Ingénieur Chercheur CNES et enfin Julien Malik, Ingénieur C-S Communication & Systèmes, pour avoir suivi avec intérêt mes travaux de recherche lors de nos nombreuses réunions et pour la bienveillance dont ils ont fait preuve à mon égard. La multitude des points de vue a fortement contribué à la qualité de mes recherches et j'en ai tiré un immense bénéfice.

Je tiens à remercier le Centre National d'Études Spatiales et l'entreprise C-S Communication & Systèmes pour le cofinancement de cette thèse et leur contribution à l'effort de recherche.

Je tiens également à remercier les deux rapporteurs : M. Sébastien Lefèvre, Professeur à l'Université Bretagne-Sud et M. Grégoire Mercier, Professeur à Télécom Bretagne, pour avoir étudié et commenté avec pertinence mes travaux de recherche.

J'ai eu la chance d'effectuer ce travail de recherche dans un evironnement international en participant au projet Toloméo "Tools for Open Multi-Risk Assessment using Earth Observation Data" où j'ai séjourné 1 an au Brésil à Rio De Janeiro. J'ai connu une expérience multiculturelle très enrichissante et je tiens à remercier tous les membres du projet Toloméo présents à Rio De Janeiro pour leur accueil et leur bienveillance à mon égard. Revenant à la France, je tiens à remercier l'ensemble des équipes du laboratoire CESBIO, en particulier Julien Osman et Mohamed Kadiri pour leur amitié et nos échanges divers sur ce monde et enfin Tristan Grégoire pour nos échanges "Geek" sur Tikz , Python, C++ et Git.

Je tiens à remercier infiniment ma famille pour leur soutien et leur confiance à chaque instant de ma vie qui font ce que je suis devenu aujourd'hui. Enfin, je tiens à remercier mon rayon de soleil Fiorella pour son soutien et sa patience de ces deux dernières années.

Introduction générale

Contexte

Depuis les années 1970, la télédétection a permis d'améliorer l'analyse de la surface de la Terre grâce aux images satellites produites sous format numérique (LANDSAT-I en 1972). En comparaison avec les images aéroportées, les images satellites apportent plus d'information car elles ont une couverture spatiale plus importante et une période de revisite courte. L'essor de la télédétection a été accompagné de l'émergence des ordinateurs. Les ordinateurs ont permis aux utilisateurs de la communauté de la télédétection d'analyser les images satellites avec l'aide de chaînes de traitement automatiques.

La télédétection permet de mesurer l'impact de l'activité humaine sur l'environnement. En effet, cette activité peut avoir des conséquences sur le climat, le terrain avec le développement des cultures dans le mileu agricole et l'urbanisation galopante, *etc.* Grâce à la télédétection, nous pouvons déterminer la composition des sols, catégoriser les types de végétation sur un territoire, calculer les indices tels que l'évaporation et la biomasse et cartographier les conséquences des activités humaines comme l'urbanisation et l'utilisation de certains pesticides. Pour effectuer cela, il est nécessaire de produire des cartes d'occupation des sols qui représentent une cartographie de types homogènes de milieux (zones urbaines, zones agricoles, forêts, véhicule, arbre,...) de la surface des terres émergées. La production de ces cartes est effectuée à l'aide de chaînes de traitement automatiques utilisant des méthodes de segmentation pour l'extraction de données et de classification pour l'identification des objets dans une scène satellite.

Ce nouveau dynamisme a aussi eu des effets positifs du point de vue technologique. Depuis les années 1970, les différentes missions d'observation de la Terre ont permis d'accumuler une quantité d'information importante dans le temps. Ceci est dû notamment à l'amélioration du temps de revisite des satellites pour une même région, au raffinement de la résolution spatiale et à l'augmentation de la fauchée (couverture spatiale d'une acquisition). La télédétection, autrefois cantonnée à l'étude d'une seule image, s'est progressivement tournée vers l'analyse de longues séries d'images multispectrales acquises à différentes dates. Une illustration de ce changement est le programme Sentinel de l'ESA, avec notamment la mission Sentinel-2 [1], qui fournira une couverture complète des terres émergées tous les 5 jours avec 13 bandes spectrales et des résolutions spatiales de 10, 20 et 60 mètres. Cela correspondra potentiellement à un volume de 1,8 téraoctets de données par jour.

En parallèle, la performance des processeurs des ordinateurs s'est aussi accrue. La loi de Moore [2] prédisait que le nombre de transistors dans les microprocesseurs doublerait tous les deux ans, ce qui a été plus ou moins le cas jusqu'à présent. Cependant, la capacité des mémoires internes [1] n'a pas suivi l'évolution de la puissance de calcul des ordinateurs [3]. La situation actuelle montre que les ordinateurs ne sont capables de garder qu'une faible proportion des volumes de données à exploiter en mémoire interne. Par conséquent, durant un traitement, une application nécessite d'accéder fréquemment

1. La mémoire interne est la mémoire pour laquelle le CPU ("Control Processor Unit") peut accéder aux données sans utiliser le bus entrée/sortie ("IO channels").

à la mémoire externe [2], ce qui constitue le principal goulot d'étranglement à cause des CPU qui attendent le rapatriement des données depuis la mémoire externe en mémoire interne. En effet, le temps d'accès en mémoire externe est 1000 à 1000000 fois supérieur à celui de la mémoire interne, ce qui entraîne des temps de latence importants.

En télédétection, l'augmentation du volume de données à exploiter est devenue une problématique due à la contrainte de la mémoire [4]. Les algorithmes de télédétection traditionnels ont été conçus pour des données pouvant être stockées en mémoire interne tout au long du traitement. Cette condition ne sera plus vraie lors du traitement d'images à très hautes résolutions spatiale, spectrale et temporelle comme celles fournies par la constellation des satellites Pléiades [5] ou par les satellites Sentinel-2. Les algorithmes de télédétection traditionnels nécessitent d'être revus et adaptés pour le traitement de données à grande échelle. Ce besoin n'est pas propre à la télédétection et se retrouve dans d'autres secteurs comme le web, la médecine, la reconnaissance vocale, *etc.*

Cette thèse se focalise sur l'adaptation des algorithmes de télédétection pour le traitement de volumes de données massifs qui ne peuvent être stockés en mémoire interne.

Besoins

Face à la problématique du traitement de larges volumes de données liée à la contrainte de la mémoire, il est nécessaire d'apporter des solutions pour adapter les algorithmes de télédétection traditionnels. En particulier, l'étude sera faite sur les méthodes de segmentation et de classification, qui constituent les étapes les plus importantes dans le processus d'extraction d'information en télédétection.

Afin de préparer les futures missions d'observation de la Terre, il est nécessaire d'adapter ces algorithmes pour le traitement de volumes de données ne pouvant être stockés en mémoire tout en garantissant des résultats identiques à ceux que nous aurions obtenus dans le cas où la mémoire ne serait pas une contrainte. Répondre à ce besoin constitue un réel enjeu pour la construction de chaînes de production des cartes d'occupation des sols à l'échelle globale.

Segmentation

Elle constitue une étape fondamentale pour l'extraction d'objets d'intérêt dans l'image. Les récentes missions d'observation de la Terre fournissent des images multi-spectrales à très haute résolution avec une couverture spatiale importante. La taille de ces images peut-être conséquente. Par exemple, la constellation des satellites Pléiades [5] fournit quotidiennement des images de résolution 70 cm rééchantillonées à 50 cm avec une couverture spatiale de 20×20 km^2. La taille standard d'une image Pléiades est donc de 40000×40000 pixels. Une description plus détaillée des satellites Pléiades est faite dans l'annexe A. Une telle image requiert quelques gigaoctets pour être stockée en mémoire.

2. La mémoire externe est la mémoire pour laquelle le CPU accède aux données en utilisant le bus entrée/sortie

La segmentation de celle-ci fait intervenir l'utilisation de structures de données nécessitant une quantité de mémoire égale voire beaucoup plus importante que celle nécessaire pour stocker l'image. Par conséquent, il peut s'avérer impossible de segmenter une telle image en une seule fois à cause de la limitation de la mémoire.

Face à cette situation, la stratégie généralement choisie est de diviser l'image en tuiles [3], de segmenter chaque tuile séparément et de reconstituer l'image segmentée en regroupant les tuiles [6]. Bien que cette stratégie résolve le problème de la mémoire, nous verrons qu'elle ne garantit pas un résultat identique à celui obtenu sans tuilage. Ceci est problématique dans la mesure où la qualité de l'identification des objets d'intérêt dans une scène pourrait être dégradée.

Ainsi, une méthode revisitant la façon de découper les tuiles afin d'assurer un résultat de segmentation identique à celui obtenu sans tuilage doit être proposée pour assurer la qualité des futures cartes d'occupation des sols.

Classification

La classification permet de catégoriser les objets présents dans la scène d'une image satellite. Elle est souvent composée d'une phase d'apprentissage qui permet de construire une fonction de décision à partir d'exemples déjà caractérisés. Ces exemples constituent un ensemble de données d'apprentissage où chaque donnée est un vecteur d'attributs et est associée avec une étiquette qui représente une classe (forêt, zone agricole, bâtiment,...). L'augmentation de la fréquence de revisite d'une même région de la Terre induit une augmentation de la taille des ensembles de données d'apprentissage.

Généralement, un algorithme d'apprentissage est complexe en temps de calcul puisqu'il nécessite souvent de nombreuses itérations. Par conséquent, lorsque l'ensemble des données en entrée devient trop volumineux pour être stocké en mémoire interne, cela peut nécessiter de nombreux accès en lecture et écriture dans la mémoire externe. Ces nombreux accès peuvent rendre les algorithmes inutilisables lors du traitement de données à grande échelle à cause du temps de latence qui est important.

Ainsi, des solutions algorithmiques doivent être investiguées afin d'utiliser des structures de données optimales pour assurer une visibilité totale de l'algorithme d'apprentissage sur l'ensemble des données, tout en minimisant les accès dans la mémoire externe.

Cadre de la thèse

Cette thèse se situe dans le cadre de l'élaboration de chaînes de traitement automatiques de production des cartes d'occupation des sols à partir d'images à hautes ou très hautes résolutions spatiale, temporelle et spectrale afin de préparer l'exploitation des données issues des missions spatiales d'observation de la Terre comme les satellites Pléiades et la famille des satellites Sentinel.

3. Une tuile représente une portion rectangulaire de l'image dont les côtés sont parallèles à ceux de l'image.

Les méthodes développées dans cette thèse ont été validées à partir d'images THR (Très Haute Résolution) fournies par les satellites Pléiades et des séries temporelles d'images multi-spectrales fournies par le satellite Landsat-8 et le satellite SPOT 4 lors de l'experience SPOT 4 (Take 5). Cette expérience a pour but de simuler les futures données Sentinel-2.

Les solutions proposées dans cette thèse ont donc pour objectif de lever un verrou scientifique lié à l'exploitation de grands volumes de données. Elles ont pour objectif d'application une meilleure compréhension des phénomènes imagés par télédétection et l'ouverture de nouvelles perspectives sur l'interprétation des données satellitaires.

Plan

Partie I

La première partie constitue une introduction aux algorithmes et aux différentes techniques qui permettent de les adapter pour le traitement de données qui ne peuvent être stockées dans la mémoire interne de l'ordinateur. Dans un premier temps, nous définissons un algorithme et nous décrivons comment il est retranscrit en langage machine pour être exécuté par un ordinateur. Nous introduisons ensuite l'analyse de la complexité d'un algorithme qui permet d'étudier son efficacité et sa performance. En particulier, nous examinons comment le choix des structures de données peut influencer la complexité d'un algorithme. Nous définissons ensuite la notion d'algorithme échelonnable et nous étudions les différentes techniques de passage à l'échelle proposées pour le traitement de grands volumes de données. Enfin, une définition de la stabilité d'un algorithme échelonnable est proposée et nous introduisons une méthode permettant de la valider.

Partie II

La seconde partie s'attache à l'étude du passage à l'échelle des méthodes de segmentation en télédétection. Le Chapitre 2 effectue une revue globale des méthodes de segmentation. Nous y expliquons, de manière plus détaillée, le fonctionnement des méthodes de segmentation par fusion de régions. Le Chapitre 3 décrit le nouvel algorithme générique de fusion de régions, baptisé algorithme GRM ("Generic Region Merging"). Cet algorithme est développé, dans le cadre de cette thèse, pour unifier la famille des algorithmes de segmentation par fusion de régions. Le Chapitre 4 décrit le processus de stabilisation de l'algorithme GRM pour la segmentation d'images trop volumineuses, qui ne peuvent être stockées dans la mémoire interne. Le Chapitre 5 décrit la version échelonnable de l'algorithme GRM, baptisé algorithme LSGRM ("Large Scale Generic Region Merging"). Cet algorithme permet de segmenter des images de taille arbitraire tout en garantissant un résultat identique à celui obtenu si la contrainte mémoire n'existait pas. Enfin, le Chapitre 6 valide la stabilité de l'algorithme LSGRM et montre la faisabilité de celui-ci par la segmentation de plusieurs scènes Pléiades à très haute résolution.

Partie III

La troisième partie s'attache à l'étude du passage à l'échelle des méthodes d'apprentissage supervisé en télédétection. Le Chapitre 8 effectue une revue des méthodes d'apprentissage existantes et justifie le choix de l'algorithme des forêts aléatoires comme algorithme d'étude pour ces travaux. Le Chapitre 9 décrit l'algorithme des forêts aléatoires et identifie les points critiques concernant la limitation de la mémoire. Le Chapitre 10 propose une version échelonnable de l'algorithme des forêts aléatoires, baptisée algorithme SMART ("Scalable Multi strAtegy Random Trees"). Cet algorithme est conçu pour assurer un classifieur identique à celui obtenu si la contrainte mémoire n'existait pas. Enfin, le Chapitre 11 valide expérimentalement la stabilité de l'algorithme SMART et montre sa faisabilité par l'apprentissage supervisé sur des volumes de données ne pouvant être stockés en mémoire interne tout au long du traitement.

Introduction aux algorithmes échelonnables

Chapitre 1

Introduction aux algorithmes échelonnables

Objectifs du chapitre

Ce premier chapitre introduit les bases de notre futur raisonnement par une exploration des différentes techniques algorithmiques afin de pallier la limitation de la mémoire. Dans un premier temps, la définition d'un algorithme et la notion de passage à l'échelle sont introduites. Les différents types de mémoire disponibles dans un ordinateur sont ensuite classifiés suivant le temps d'accès et la capacité de stockage. À partir de cette classification, nous décrivons une modélisation simplifiée de l'ordinateur que nous considérons tout au long de nos travaux. Afin d'analyser l'usage de la mémoire et le temps d'exécution d'un algorithme, nous étudions la complexité de celui-ci avec la notation de Landau. Elle permet de mesurer l'efficacité et la performance d'un algorithme dans le cas le plus défavorable. Une des étapes fondamentales pour rendre un algorithme efficace et performant, c'est-à dire minimisant à la fois l'usage de la mémoire et le nombre d'opérations, est le choix judicieux des structures de données utilisées au cours du traitement. Nous verrons ainsi, à travers un exemple, comment le choix d'une structure de données peut affecter la complexité d'un algorithme. Enfin, nous présentons les différentes techniques de passage à l'échelle existantes pour adapter un algorithme au traitement de données ne pouvant être stockées dans la mémoire interne de l'ordinateur. Dans la suite de ce manuscrit, un algorithme capable de traiter des volumes de données de taille arbitraire est qualifié d'algorithme échelonnable.

1.1 Définition et notations

1.1.1 Définition d'un algorithme

La notion d'algorithme fait référence à une procédure pour résoudre un problème posé. Un célèbre exemple d'algorithme est celui d'Euclide qui consiste à déterminer le plus grand diviseur commun de 2 entiers positifs (appelé communément le PGCD). Sa procédure est illustrée par la figure 1.1.

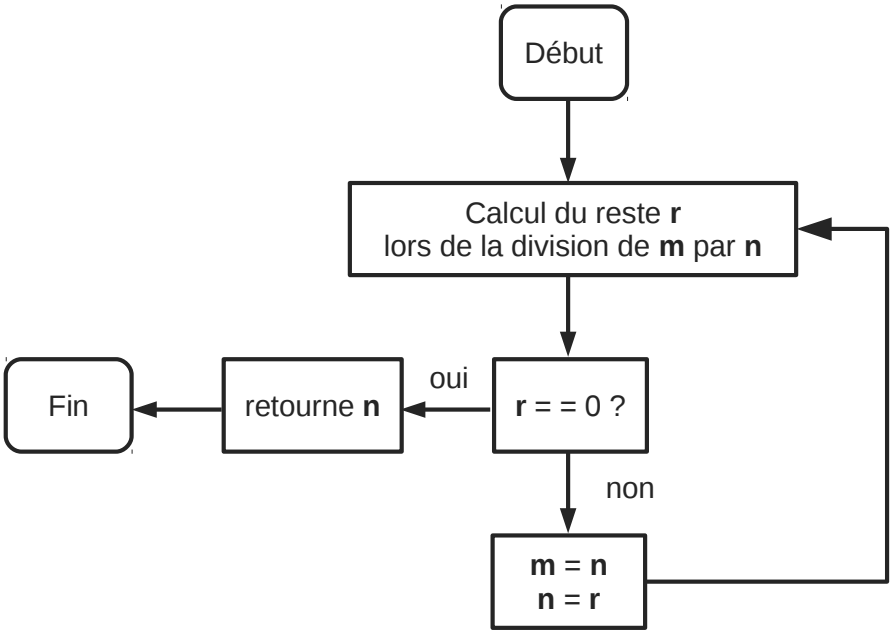


Figure 1.1 – Algorithme d'Euclide pour le calcul du PGCD.

Cet algorithme est composé de 3 étapes qui consistent à effectuer une division, une comparaison et enfin de nouvelles affectations. L'algorithme d'Euclide tel qu'il est décrit est caractérisé par :

- Le type des données qu'il reçoit en entrée : 2 entiers positifs.
- Une suite finie et ordonnée d'opérations réalisables pour aboutir à la solution du problème.
- Le type des données qu'il retourne : 1 entier positif.

Nous proposons ainsi la définition suivante pour un algorithme :

Definition Un algorithme est une suite finie et ordonnée d'opérations réalisables qui permet de résoudre un problème donné à partir d'un ensemble de données d'un certain type.

1.1.2 Passage à l'échelle

Le terme anglais "Scalability" désigne la capacité d'un algorithme à traiter avec la même efficacité des volumes de données de taille arbitraire. Généralement, le temps d'exécution de tels algorithmes évolue linéairement en fonction de la taille des données en entrée. En français, plusieurs traductions sont possibles pour l'adjectif "scalable" : évolutif, extensible, faisable, échelonnable, *etc.* Nous avons choisi l'adjectif échelonnable car sa signification est en adéquation avec les objectifs de cette thèse. Nous utilisons le terme échelonnable pour décrire un algorithme capable de traiter des volumes de données de taille arbitraire tout en maintenant un temps d'exécution linéaire ou linéarithmique et en assurant des résultats identiques à ceux obtenus dans le cas où les données peuvent être stockées en mémoire. L'action consistant à développer un algorithme échelonnable à partir d'un algorithme classique est définie par le terme de passage à l'échelle dans la suite de ce manuscrit.
En télédétection, certains algorithmes de segmentation et de classification ne peuvent être appliqués lorsque la taille des données dépasse la capacité de stockage de la mémoire interne de l'ordinateur et cela pour plusieurs raisons. La première raison est le manque de visibilité des données lorsque celles-ci ne peuvent être stockées en mémoire interne. L'algorithme n'a alors qu'une visibilité partielle des données, qui peut ne pas être représentative de l'ensemble des données initial. Ce manque d'information peut alors affecter la qualité des résultats. La seconde raison est la nécessité d'effectuer de nombreux accès en lecture et écriture dans la mémoire externe de l'ordinateur. Le temps d'exécution de l'algorithme peut devenir excessivement long car ces opérations sont coûteuses à cause du temps de latence important. Cela peut ainsi compromettre l'utilisation de l'algorithme, surtout dans des domaines où la performance est requise (applications temps réel). Ces algorithmes ne sont donc pas échelonnables au sens où nous l'avons défini.
L'objectif principal de cette thèse est d'étudier le passage à l'échelle des algorithmes de segmentation et de classification tout en répondant au même problème et en obtenant des résultats identiques à ceux obtenus dans le cas où la mémoire ne serait pas limitée.
Avant de présenter les techniques de passage à l'échelle, il est nécessaire de comprendre comment un algorithme est retranscrit en un programme informatique et comment celui-ci utilise les ressources de l'ordinateur.

1.1.3 Algorithmes et programmation informatique

L'apparition des ordinateurs a permis aux scientifiques d'appliquer des algorithmes sur des données de manière automatique. L'enjeu constitue le développement de programmes informatiques qui permettent à l'ordinateur d'appliquer les différentes étapes de l'algorithme sur des données en entrée. Cette technique est nommée la programmation informatique. Pour réaliser un programme informatique, des langages de programmation sont apparus pour aider l'homme à retranscrire un problème posé avec le langage naturel en un problème compréhensible par l'ordinateur. Nous pouvons citer quelques langages célèbres de programmation : C [7], C++ [8], Python [9], *etc.* Ces langages ont un vo-

cabulaire proche de celui du langage naturel. Ils offrent une abstraction de certaines opérations algorithmiques comme les opérations arithmétiques, les boucles mais aussi une manière simplifiée de gérer l'utilisation de la mémoire par le processeur. Ces langages permettent aussi de modéliser la façon dont les données sont organisées dans la mémoire d'un ordinateur pour ensuite être traitées de la manière la plus efficace possible par le programme informatique. Les différentes manières pour concevoir un programme informatique, à partir d'un algorithme énoncé dans le langage naturel, sont regroupées dans le domaine des techniques algorithmiques.

Avant d'étudier la conception de nouveaux algorithmes pour le traitement de grands volumes de données en télédétection, il est nécessaire de présenter les techniques algorithmiques existantes pour constituer les bases de notre raisonnement. Concevoir un programme informatique nécessite de connaître la syntaxe et la grammaire du langage de programmation utilisé. Pour les travaux de cette thèse, le langage utilisé est le C++ car il permet d'avoir un contrôle total sur la performance de l'algorithme et sur l'usage de la mémoire. La raison secondaire est que l'ensemble des algorithmes développés dans cette thèse seront intégrés dans le logiciel de traitement d'images Orfeo Toolbox développé par le CNES, qui est lui-même écrit en C++. Concevoir un programme informatique nécessite de connaître l'architecture de l'ordinateur sur lequel le programme est exécuté. Une connaissance de la capacité de stockage des différents types de mémoire ainsi que leur hiérarchie permet de concevoir des programmes tirant bénéfice au maximum de la localité des données. Ceci s'effectue en partie à travers le choix des structures de données qui jouent un rôle fondamental dans l'efficacité d'un traitement. Concevoir un programme informatique efficace et performant passe également par une connaissance algorithmique approfondie. C'est pourquoi, l'ensemble des techniques algorithmiques proposées dans cette thèse s'inspire de la lecture des deux ouvrages : *The Art Of Computer Programming* [10] et le célèbre CLRS (*Introduction to Algorithms*) [11], qui constituent des références dans ce domaine. L'ensemble des algorithmes décrits dans cette thèse seront illustrés soit par un diagramme d'exécution soit par un pseudo-code indépendant du langage de programmation.

1.2 Modélisation d'un ordinateur

1.2.1 Description générale d'un ordinateur

Modéliser la pensée humaine et son mode de fonctionnement pour aboutir à une intelligence artificielle est le rêve de nombreux chercheurs. Même si cet objectif n'a pas encore été atteint, une partie du fonctionnement de la pensée humaine a pu être élaborée : la capacité à calculer. La création de machines capables d'effectuer des calculs a été l'objectif de nombreux scientifiques par le passé. Les premières machines à calculer étaient mécaniques. Elles étaient construites en bois et en métal. Nous pouvons citer quelques pionners dans ce domaine tels que Leibniz ou Charles Babage (1791-1871). Ce dernier a inventé la machine différentielle (1821) puis la machine analytique où il était

possible pour l'homme de programmer des calculs par le moyen de cartes perforées. Cependant, à cette époque, il n'existait pas encore une définition formelle d'une fonction calculable. Il a fallu attendre 1936 pour la première modélisation abstraite d'un ordinateur avec Alan Turing [12]. Dans son article, Alan Turing décrit un ordinateur avec un automate caractérisé par des configurations et un ruban où l'automate vient y lire, écrire et modifier des symboles. À chaque configuration est associée une ou plusieurs opérations qui consistent à lire ou écrire des symboles sur le ruban supposé infini. L'ensemble de ces opérations conduit l'automate à se trouver dans une nouvelle configuration.
L'ordinateur d'aujourd'hui est très similaire à la machine à calculer proposée par Alan Turing. Dans cette thèse, nous modélisons un ordinateur par une unité de calcul (automate) exécutant des instructions sur les données de manière séquentielle. Ces instructions constituent des opérations sur des séquences de bits (symboles). La performance de l'unité de calcul est déterminée par le nombre d'opérations qu'elle peut effectuer en une unité de temps. Une unité de calcul dispose d'un registre pour stocker les résultats intermédiaires entre deux opérations (ruban). Le temps d'accès aux données dans un registre s'effectue en un cycle d'horloge processeur. Cependant, la capacité de stockage du registre est très limitée. Ainsi, pour pouvoir traiter une donnée, il est nécessaire de la rapatrier depuis la mémoire de l'ordinateur. Ce temps de latence est souvent la cause du manque d'efficacité d'un traitement [1]. Contrairement au ruban infini, la mémoire est limitée dans un ordinateur, ce qui contraint à une visibilité réduite de l'ensemble des données si celui-ci ne peut être stocké en mémoire. Cette problématique sur la visibilité des données constitue l'enjeu majeur de cette thèse. Dans la section suivante, nous listons les différents types de mémoire présents dans un ordinateur et présentons leurs caractéristiques.

1.2.2 Différents types de mémoire

Plusieurs types de mémoire sont disponibles dans un ordinateur standard. Chacune d'elles se caractérise par son temps de latence et sa capacité de stockage. Les mémoires peuvent être classifiées en 2 catégories principales : les mémoires vives et les mémoires mortes.

Les mémoires vives

Une mémoire vive permet de stocker les données tant que celle-ci est alimentée par un courant électrique. Une rupture de l'alimentation entraîne la perte des données. Ce type de mémoire se caractérise par un accès aux données rapide et une faible capacité de stockage. Elle est généralement située physiquement près de l'unité de calcul. Augmenter la capacité de ce type de mémoire est coûteux et ne peut se faire indéfiniment. Il existe 3 types de mémoire vive qui sont listés ci-dessous par capacité de stockage croissante et par vitesse d'accès décroissante :

1. "All computers wait at the same speed", Wernher von Braun.

- Le registre. C'est la mémoire utilisée par l'unité de calcul du processeur pour stocker les résultats intermédiaires d'un calcul ainsi que les instructions. La capacité des registres actuels est de l'ordre de quelques milliers d'octets. Le temps de latence est inexistant car les données et les instructions sont récupérées en 1 cycle d'horloge du processeur.

- La mémoire cache. C'est la mémoire tampon utilisée pour stocker les données et les instructions qui sont fréquemment utilisées par l'unité de calcul. Elle permet d'amortir le temps de latence dû aux accès à la mémoire centrale. Selon l'architecture de l'ordinateur, il peut exister une hiérarchie de mémoires cache. Sur les ordinateurs actuels, cette hiérarchie est généralement composée de 3 niveaux de cache notés L1, L2 et L3 avec des capacités de stockage croissantes et des vitesses d'accès décroissantes. La capacité globale de l'ensemble des mémoires cache est de l'ordre de quelques mégaoctets. Le temps de latence est de l'ordre de quelques nanosecondes.

- La mémoire centrale, également appelée mémoire RAM (Random Access Memory), est utilisée par l'ordinateur pour stocker les données et les instructions au cours d'un traitement. Sa capacité de stockage est de l'ordre de quelques dizaines de gigaoctets et son temps de latence est de l'ordre de quelques centaines de nanosecondes.

Dans le cadre de cette thèse, l'ensemble de ces mémoires est regroupé sous le nom de mémoire interne de l'ordinateur. Par la suite, nous simplifions en supposant que le temps de latence pour accéder aux données en mémoire interne est égal à un cycle de l'horloge du processeur. Nous verrons que cette simplification n'enlève pas de généralité aux solutions proposées car elles pourront être adaptées à de nouvelles hiérarchies mémoire par récursion.

Les mémoires mortes

La mémoire morte permet de stocker les données et de les garder même lorsque celle-ci n'est plus alimentée. Cette mémoire se caractérise par sa très grande capacité de stockage, mais avec l'inconvénient d'un temps d'accès très long comparé aux mémoires vives. Elle regroupe toutes les mémoires de masse comme les disques durs, les mémoires flash, les disques optiques et les disques SSD ("Solid State Drive"). Un avantage de ces mémoires est qu'elles sont bon marché comparées aux mémoires vives. Par conséquent, nous pouvons étendre presque à l'infini la capacité de stockage de ce type de mémoire pour un ordinateur. L'inconvénient est le temps de latence qui est de l'ordre de quelques centaines de millisecondes soit 10^6 fois supérieur à celui des mémoires vives. La raison de ce long temps de latence s'explique par l'architecture d'un disque dur illustrée dans la figure 1.2. Les données sont stockées sur la surface magnétique d'un disque qui est en rotation. La vitesse de rotation de ces disques varie aujourd'hui entre 5000 et 15000 rotations par minute (RPM). Les données sont accédées et modifiées par l'intermédiaire d'une tête de lecture qui se déplace de manière concentrique le long de la surface magnétique. Ainsi, pour accéder à un segment sur la surface, la tête de lecture doit se déplacer horizontalement dans un premier temps pour se situer sur le cercle du segment et ensuite attendre que la rotation du disque la place au dessus du segment. Le temps de latence cumulé pour effectuer cette opération sur un disque moderne est de l'ordre de 3 à 10 ms.

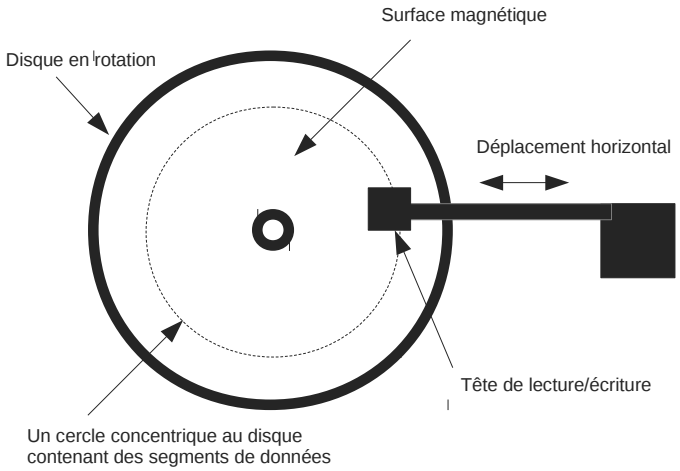


Figure 1.2 – Fonctionnement d'un disque dur.

Ces mémoires sont regroupées sous le nom de mémoire externe et nous considérons que la quantité de celle-ci pour stocker les données en entrée d'un algorithme est toujours suffisante.

Pour résumer, la table 1.1 présente la structure hiérarchique des types de mémoire avec leurs temps d'accès mesurés sur un ordinateur avec un processeur de 3 GHz. Les données sont issues de la présentation faite lors de la conférence CppCon et sont disponibles dans la présentation "Efficiency with Algorithms, Performance with Data Structures" [2].

Type de mémoire	Temps de latence
1 cyle d'horloge	1 ns
cache L1	0.5 ns
cache L2	7 ns
mémoire centrale	10^2 ns
SSD	$1,5 \times 10^5$ ns
Disque dur	10^7 ns

Table 1.1 – Hiérarchie mémoire et temps de latence en nanosecondes (ns).

1.2.3 Modélisation d'un ordinateur dans le cadre de nos travaux

Pour la conception de nouveaux algorithmes de segmentation et de classification, nous nous sommes basés sur un modèle générique d'un ordinateur qui est illustré par la figure 1.3. Un ordinateur est donc composé d'un processeur, d'une mémoire interne et d'une mémoire externe. Les données peuvent-être transférées entre le processeur et la mémoire interne et entre la mémoire externe et la mémoire interne.

1.3 Complexité algorithmique

L'analyse de la complexité d'un algorithme étudie l'évolution du nombre d'opérations élémentaires de celui-ci de façon aymptotique. Nous analysons également la complexité en espace de l'algorithme, qui traduit la quantité de mémoire utilisée au cours du traitement de façon asymptotique. Nous parlons ainsi de complexités en temps et en espace de l'algorithme. Une opération élémentaire est une opération qui est exécutée dans un temps constant par le processeur. Ces opérations élémentaires regroupent les opérations arithmétiques telles que les additions, les soustractions, les multiplications, mais également les opérations de comparaison et les opérations d'affectation. En notant n la taille des données en entrée, l'algorithme requiert un certain nombre d'opérations élémentaires ainsi qu'une certaine quantité de mémoire pour le traitement. Nous notons $c_t(n)$ le coût

2. https://github.com/CppCon/CppCon2014

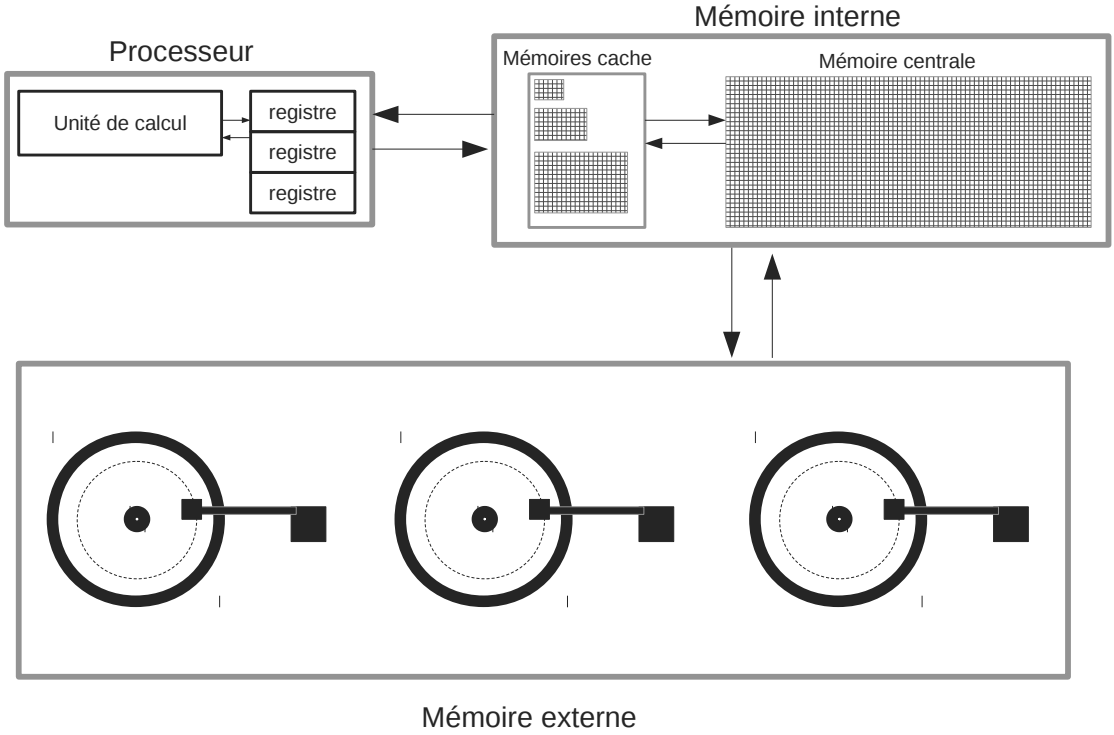


Figure 1.3 – Modélisation simplifiée d'un ordinateur considérée dans le cadre de cette thèse.

en temps pour effectuer ces opérations et $c_m(n)$ le coût en espace.
Par exemple, considérons l'algorithme 1.1 qui permet de trouver la valeur maximum dans un tableau T de taille n et déterminons $c_t(n)$ et $c_m(n)$.

1: **procédure** TROUVEMAXIMUM($T = [a_1, a_2, ..., a_n]$)
2: $max = T[0]$
3: **pour** $i = 0$; $i <$taille(T) ; $i \leftarrow i + 1$ **exécute**
4: **si** $max < T[i]$ **alors**
5: $max = T[i]$
6: **fin si**
7: **fin pour**
8: **retourne** max
9: **fin procédure**

Algorithme 1.1 – Algorithme pour retrouver le nombre maximum dans un tableau de valeurs.

Cet algorithme requiert de stocker le tableau T de taille n, ainsi que la valeur maximum du tableau max, c'est-à-dire $n + 1$ éléments. Ainsi $c_m(n) = n + 1$. Dans la ligne

2 de l'algorithme 1.1, un accès à la valeur dans T à la position 0 puis une affectation sont effectuées. Ces 2 actions constituent 2 opérations élémentaires. Avant le début de la boucle (ligne 3), 2 opérations élémentaires sont effectuées à savoir $i = 0$ et $i < \text{taille}(T)$. Pour chaque itération de la boucle, 2 opérations élémentaires sont effectuées, à savoir l'incrémentation de i et la comparaison $i < \text{taille}(T)$. Dans le contenu de la boucle, au maximum 4 opérations élémentaires sont réalisées à savoir l'accès à la valeur $T[i]$, la comparaison de celle-ci avec max et si $max < T[i]$ alors l'accès de nouveau à $T[i]$ et son affectation à max. Le coût total en temps d'exécution est $c_t(n) = 6 \times n + 4$.

Nous venons de déterminer les coûts toujours dans le cas le plus défavorable. En effet, lors de l'analyse d'un algorithme, nous considérerons toujours le pire des cas afin d'obtenir une limite supérieure sur le temps d'exécution et sur la quantité de mémoire nécessaire.

1.3.1 Notation de Landau

La notation de Landau ([10] Chapitre 1 Section 2.11) permet d'évaluer l'efficacité d'un algorithme en donnant un ordre de grandeur du coût pour le temps d'exécution et pour l'espace mémoire nécessaire en fonction de la taille des données en entrée de l'algorithme. Il existe plusieurs notations de Landau et nous considérons seulement celle correspondant à une majoration des coûts représentant ainsi le cas le plus défavorable. Elle est notée O et se définit formellement de la façon suivante. Supposons 2 fonctions f et g définies sur $\mathbb{R}$, nous avons :

$$f = O(g(x)) \text{ lorsque } x \to \infty \Leftrightarrow \exists M, x_0 \in \mathbb{R}^* \text{ tel que } \frac{|f(x)|}{|g(x)|} \leq M, \forall x \geq x_0$$

Cela signifie que les comportements de $f(x)$ et $g(x)$ sont les mêmes à une constante près lorsque $x \to \infty$.
Concernant l'algorithme de notre exemple (Algorithme 1.1), nous avons déterminé que $c_t(n) = 6 \times n + 4$. Supposons la fonction $g(n) = n$, le ratio entre $c_t(n)$ et $g(n)$ lorsque $n \to \infty$ vaut :

$$\frac{c_t(n)}{g(n)} = \frac{6 \times n + 4}{n} \approx 6 \text{ lorsque } n \to \infty$$

Cela signifie qu'à une constante près, le coût d'exécution en temps $c_t(n)$ de notre algorithme est de l'ordre de grandeur de $g(n)$. La notation de Landau de l'algorithme est notée $O(n)$ et nous disons qu'il a une complexité en temps d'exécution linéaire. Similairement, la notation de Landau pour $c_m(n)$ est $O(n)$ et nous disons qu'il a une complexité en espace linéaire.

Nous venons de voir que la notation de Landau permet de donner un ordre de grandeur du comportement général d'un algorithme lorsque la taille des données en entrée tend vers l'infini. La complexité en espace est devenue, avec le traitement de grands volumes de données, aussi essentielle que celle liée au temps d'exécution et c'est pourquoi nous y attachons une grande importance dans nos travaux. En effet, il est très fréquent

qu'un algorithme utilise des structures pour stocker des données au cours de la procédure. Un espace mémoire supplémentaire est souvent requis en plus de celui nécessaire pour stocker les données en entrée.

La liste suivante énumère les complexités communes avec la notation de Landau de la plus avantageuse à la plus problématique concernant le caractère échelonnable de l'algorithme :

- $O(1)$: complexité constante. Elle ne dépend pas de la taille des données en entrée.
- $O(\log n)$: complexité logarithmique. C'est en général la complexité en temps pour parcourir les structures arborescentes.
- $O(n)$: complexité linéaire. La complexité est proportionnelle à la taille des données en entrée.
- $O(n \log n)$: complexité linéarithmique. C'est en général la complexité en temps d'exécution d'un algorithme qui divise récursivement l'ensemble des données en entrée en plusieurs blocs et applique un algorithme de complexité en temps linéaire sur chaque bloc. Un algorithme de tri célèbre nommé le MergeSort a une complexité $O(n \log n)$ et est décrit dans [11] Chapitre 2 Section 2.3.1.
- $O(n^2)$: complexité quadratique. La complexité est proportionnelle au carré de la taille des données en entrée. Cette complexité est particulièrement problématique pour le traitement de grands volumes de données.
- ...

La notation de Landau a certaines limitations car elle permet d'avoir seulement un ordre de grandeur du comportement d'un algorithme de façon asymptotique en considérant la pire situation. Par exemple, elle ne tient pas compte des facteurs constants. En effet, un algorithme A nécessitant $3 \times n$ instructions et un algorithme B nécessitant $20 \times n$ instructions auront la même notation de Landau $O(n)$ alors que l'algorithme A est plus rapide que B.
De plus, la notation de Landau ne tient pas compte de la répartition des données en entrée. En effet, le tri par insertion ([11] Partie I Chapitre 2 Section 1), qui a une complexité $O(n^2)$ dans le pire des cas, peut avoir un comportement linéaire si les données en entrée sont déjà plus ou moins triées et ainsi être plus performant qu'un algorithme de tri ayant une complexité $O(n \log n)$ comme le Heapsort ([11] Partie II Chapitre 6 Section 1).
Enfin, l'analyse avec la notation de Landau ne tient pas compte non plus du type de stucture de données utilisé. En effet, le parcours dans un tableau dynamique où toutes les données sont contigües en mémoire peut-être beaucoup plus rapide que le parcours d'un arbre binaire de recherche ([11] Partie III Chapitre 12 Section 1) où les données sont aléatoirement placées en mémoire interne. En effet, la localité des données est importante pour profiter de la hiérarchie des caches et nous verrons qu'en tenir compte par la suite permet d'améliorer significativement l'efficacité de l'algorithme. Il existe différents types

de structure de données (tableau, liste) qui organisent les données en mémoire de façon différente. Avec les processeurs actuels, il est parfois surprenant de voir, qu'en choisissant une structure favorisant la localité des données, les vitesses d'exécution de certains algorithmes ne sont pas celles qu'on pourrait espérer étant donné leur complexité.
C'est pourquoi, nous utilisons la notation de Landau pour avoir une idée globale du comportement de nos algorithmes. Par contre, pour des analyses plus approfondies, nous étudierons la répartition des données en entrée ainsi que les caractéristiques des différentes structures de données pour concevoir des algorithmes efficaces du point de vue de l'accès à la mémoire.

À noter que ce n'est pas seulement les algorithmes ayant une complexité en temps ou en espace quadratique qui ne sont pas échelonnables. Cela peut être également le cas pour des algorithmes de complexité linéaire. En effet, si la taille des données en entrée dépasse la quantité de mémoire interne disponible alors le manque de visibilité peut entraîner des modifications sur le résultat final. Il est donc nécessaire de trouver des solutions qui permettent de les rendre échelonnables. Une des pistes à explorer est le choix judicieux des structures de données. Nous verrons que leur rôle est fondamental dans le passage à l'échelle d'un algorithme.

1.4 Structures de données

Dans un programme informatique, une structure de données représente la façon dont sont organisées les données en mémoire.
Différents types de structures de données existent suivant les besoins de l'application. Le choix d'une structure de données intervient directement dans la complexité de l'algorithme comme nous allons le voir dans les deux exemples suivants. Le choix des structures de données constitue ainsi une étape primordiale lors de la conception d'un algorithme échelonnable.

Une structure de données célèbre est la liste chaînée. Une telle liste est composée de nœuds contenant, en plus de données spécifiques à l'application, un pointeur vers le nœud suivant noté *suiv*. Dans la suite, nous considérons que le nœud contient, en plus d'un pointeur, une valeur entière notée *valeur*. Pour un nœud, noté *nd*, l'accès à l'un de ses attributs, par exemple *valeur*, est noté *nd.valeur*. Cette syntaxe est utilisée pour tous les pseudo-codes dans ce manuscrit. Un exemple d'implémentation d'une liste est illustré par la figure 1.4. Une liste est donc caractérisée par un pointeur vers le premier nœud de la liste, noté *debut*.

Quatre opérations sont en général nécessaires lorsqu'une structure de données est utilisée par un algorithme, à savoir l'insertion, l'accès à une donnée suivant sa position, la recherche et la suppression. La complexité en espace de l'utilisation d'une liste simplement chaînée est $O(n)$ puisqu'il y a autant de nœuds que de données dans l'ensemble. Nous allons donc déterminer la complexité en temps des 3 opérations.
Insérer un élément consiste à rajouter un nœud en début de liste. Le pseudo-algorithme pour insérer un nœud en début de liste est illustré par l'algorithme 1.2.

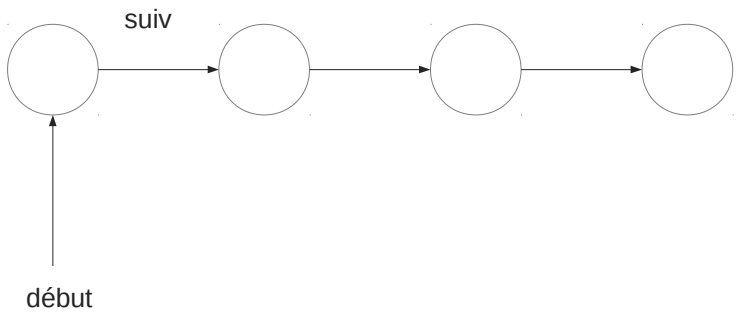


Figure 1.4 – Modélisation d'une liste simplement chaînée

1: **procédure** InsérerNoeud(nd, $debut$)
2: $nd.suiv = debut$
3: $debut = nd$
4: **fin procédure**

Algorithme 1.2 – Insertion d'un nœud en début de liste

L'opération d'insertion d'un nouveau nœud contient 2 instructions et a donc une complexité en temps constante $O(1)$.
Accéder à un nœud dans une liste étant donnée sa position i consiste à parcourir les $i - 1$ premiers nœuds et est décrit par le pseudo-algorithme 1.3. Dans le cas le plus défavorable, tous les nœuds de la liste sont parcourus, ce qui correspond au cas où le nœud est à la dernière position ou que la position est supérieure à la taille de la liste. La complexité en temps est $O(n)$.

1: **procédure** AccesNoeud($debut$, i)
2: $nd = debut$
3: $j = 0$
4: **tant que** $nd \neq NULL$ et $j < i$ **exécute**
5: $nd = nd.suiv$
6: $j++$
7: **fin tant que**
8: **retourne** nd
9: **fin procédure**

Algorithme 1.3 – Accès à un nœud étant donnée sa position i dans la liste.

Trouver un nœud dans une liste consiste à parcourir les nœuds de la liste et à comparer la valeur du nœud courant avec la valeur du nœud que nous désirons trouver. L'opération se termine lorsqu'il y a égalité ou lorsque tous les nœuds ont été visités. Le pseudo-algorithme pour trouver un nœud est illustré par l'algorithme 1.4.

Dans le cas le plus défavorable, le nœud que nous désirons trouver se trouve en fin de liste ou n'existe pas. Dans une telle situation, il est nécessaire de parcourir tous les nœuds de la liste et de comparer leur valeur avec la valeur du nœud que nous cherchons. La complexité en temps est alors $O(n)$.

```
1: procédure RechercheNoeud(valeur, debut)
2:     pour nd = debut; nd ≠ NULL; nd = nd.suiv exécute
3:         si nd.valeur == valeur alors
4:             retourne nd
5:         fin si
6:     fin pour
7:     retourne NULL
8: fin procédure
```

Algorithme 1.4 – Recherche d'un nœud dans une liste

Enfin, l'opération de suppression est dominée par l'opération de recherche du nœud à supprimer. La suppression du nœud est effectuée par la modification du chaînage du nœud précédent au nœud à supprimer. Le pseudo-code 1.5 décrit cette action. Cette dernière opération ne requiert qu'un nombre fixe d'instructions qui ne dépend pas de la taille des données en entrée. La complexité en temps de l'opération de suppression est $O(n)$.

```
1: procédure SuppressionNoeud(valeur, debut)
2:     prevNd = NULL
3:     pour nd = debut; nd ≠ NULL; nd = nd.suiv exécute
4:         si nd.valeur == valeur alors
5:             si prevNd ≠ NULL alors
6:                 prevNd.suiv = nd.suiv
7:                 retourne debut
8:             sinon
9:                 debut = nd.suiv
10:                retourne debut
11:            fin si
12:        fin si
13:        prevNd = nd
14:    fin pour
15:    retourne debut
16: fin procédure
```

Algorithme 1.5 – Suppression d'un nœud dans une liste

Ainsi, une liste simplement chaînée est une structure de données intéressante lorsque

le nombre d'opérations d'insertion en début de liste et de mise à jour du chaînage est beaucoup plus important que le nombre d'opérations de recherche et de suppression.

Considérons maintenant un tableau dynamique dont les valeurs sont stockées de manière séquentielle en mémoire. Lors de l'insertion d'un élément dans cette structure de données, l'ordre séquentiel des valeurs dans le tableau doit être maintenu dans la mémoire. Si n est la taille du tableau et que nous désirons insérer une valeur à la position i alors toutes les valeurs situées de la position i à la position n doivent être décalées d'une case vers la droite en mémoire. La figure 1.5 illustre cette opération.

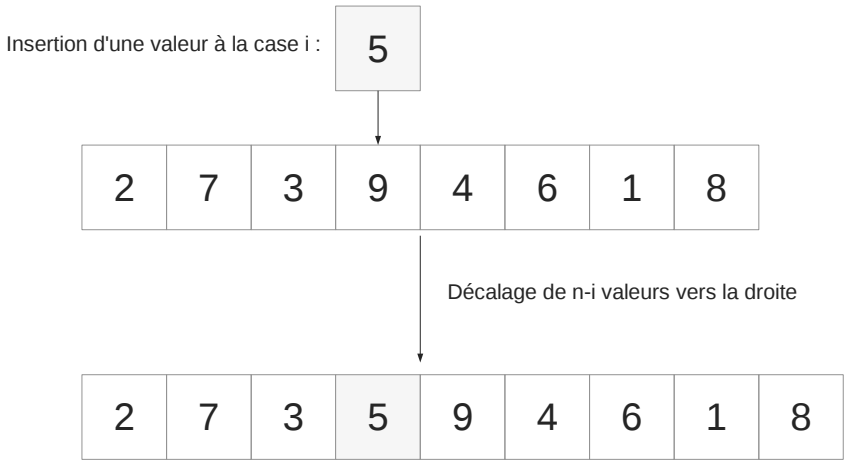


Figure 1.5 – Opération d'insertion d'une valeur à une certaine position dans la tableau dynamique

Cette opération d'insertion nécessite de redimensionner la taille du tableau dynamique et de décaler $n - i$ valeurs impliquant une réallocation de la mémoire et $n - i$ copies. Dans la pire situation, nous devons insérer la valeur en première position nécessitant de décaler n valeurs. Ainsi la complexité en temps d'une telle opération est $O(n)$. La recherche d'une valeur dans le tableau dynamique nécessite de parcourir les éléments du tableau jusqu'à trouver celui dont la valeur est égale. Dans le cas le plus défavorable, la valeur est située en dernière position du tableau ou n'est pas dans le tableau. La complexité en temps est donc $O(n)$.
De même, la suppression d'une valeur dans le tableau nécessite le parcours du tableau pour trouver la valeur et entraîne ensuite le redimensionnement et la copie des valeurs à sa droite lors de la supression pour maintenir les données contigües en mémoire interne. La complexité en temps est également $O(n)$. Notons cependant une amélioration de cet algorithme. En effet, si le nombre de données à supprimer est important, cette action peut avoir une complexité en temps quadratique, car pour chaque donnée à supprimer, il faut décaler les valeurs situées à sa droite. L'idée est alors de visiter l'ensemble des données du tableau qu'une seule fois et lorsqu'une donnée doit être supprimée, l'échanger avec la dernière valeur valide du tableau. La figure 1.6 illustre l'ensemble des opérations dans la cas où 2 valeurs doivent être supprimées d'un tableau dynamique.

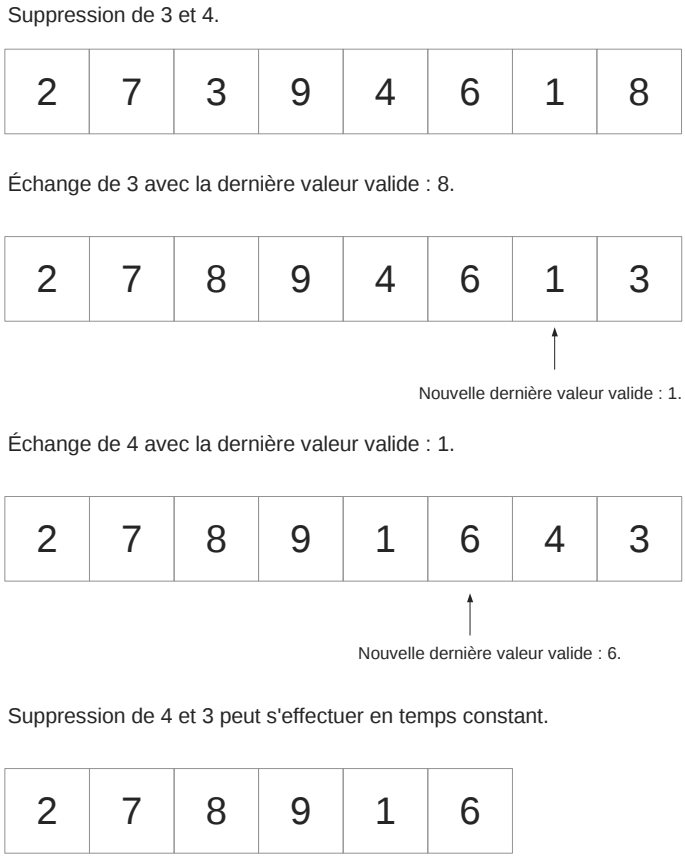


Figure 1.6 – Suppression plus efficace des valeurs dans un tableau dynamique.

En revanche, l'accès à un élément étant donné sa position est faite en un nombre fixe d'instructions. La complexité en temps est donc $O(1)$.

Nous venons de présenter 2 structures de données qui ont le même rôle mais une façon différente d'organiser les données dans la mémoire. Dans l'exemple suivant, nous allons montrer comment la complexité d'un algorithme peut être influencée par le choix des structures de données.
Supposons des polygones dans une grille à 2 dimensions composée de pixels. Chaque polygone est caractérisé par un attribut a, une liste de pixels et une liste de pointeurs vers les polygones adjacents. Deux polygones sont adjacents s'ils partagent une frontière commune. L'objectif est de fusionner les polygones selon un critère basé sur l'attribut a. Une structure de données est utilisée pour stocker les polygones de la grille et nous la notons C_p. Afin d'obtenir une croissance équilibrée des polygones tout au long de la procédure, nous choisissons de les visiter aléatoirement à chaque itération. Notons N_p le nombre de polygones restants après chaque itération. Le pseudo-code de l'algorithme pour fusionner les polygones est illustré par l'algorithme 1.6.

```
1: procédure MergePoly(C_p)
2:     tant que il reste des polygones à fusionner exécute
3:         I ← tableau aléatoire d'indices entre 0 et N_p
4:         pour chaque indice i dans I exécute
5:             si C_p[i] n'a pas déjà fusionné alors
6:                 si C_p[i] peut fusionner avec un polygone adjacent p alors
7:                     Met à jour le voisinage
8:                     Met à jour les cellules
9:                     p est marqué comme fusionné
10:                fin si
11:            fin si
12:        fin pour
13:        Supprime les polygones marqués comme fusionnés de C_p.
14:    fin tant que
15: fin procédure
```

Algorithme 1.6 – Algorithme de fusion des polygones

L'objectif est de déterminer C_p pour minimiser la complexité de l'algorithme. Pour la structure de données, nous avons le choix entre un tableau dynamique ou une liste chaînée. Dans les 2 scénarios suivants, la complexité en espace est identique et vaut $O(N_p)$. Pour simplifier nous n'étudions pas les complexités du processus de fusion des polygones et nous considérons qu'elles sont constantes $O(1)$.
Supposons dans un premier temps que C_p est une liste simplement chaînée. À chaque itération, nous visitons aléatoirement les polygones dans C_p (lignes 4 et 5). La ligne 5 vérifie si $C_p[i]$ n'a pas déjà fusionné. Si C_p est une liste, alors chercher $C_p[i]$ requiert dans le pire des cas N_p instructions pour chaque indice i. Il y a ainsi N_p^2 instructions juste pour vérifier si le polygone a fusionné. Supprimer les polygones dans une liste simplement chaînée a une complexité $O(N_p)$. Par conséquent, la complexité de l'algorithme est dominée par la boucle **tant que** : $O(N_p^2)$.
Supposons maintenant que C_p est un tableau dynamique. Trouver un polygone dans C_p étant donnée sa position s'effectue en temps constant dans un tableau dynamique. Il y a ainsi N_p instructions au maximum dans la boucle **tant que**. Enfin, la suppression des polygones qui ont fusionné est linéaire $O(N_p)$ si nous prenons la précaution de les déplacer à la fin du tableau dynamique comme cela a été expliqué dans la figure 1.6. La complexité en temps de l'algorithme est finalement $O(N_p)$ qui est bien meilleure que celle calculée pour le premier scénario. Une propriété intéressante des tableaux dynamiques est la localité des données qui sont stockées de manière contigüe en mémoire, ce qui peut ne pas être le cas pour une liste simplement chaînée. Ainsi, il faut moduler le fait qu'une structure de données soit théoriquement plus efficace qu'une autre par le temps de latence, qui dépend de la façon dont sont organisées les données en mémoire. En effet, avec les architectures mémoire actuelles, il est préférable d'utiliser des structures

de données alignées comme les tableaux dynamiques car le transfert de données entre la mémoire centrale et les mémoires cache s'effectue par bloc de données contigües. Cette analyse doit être faite de manière expérimentale en comparant les structures de données par rapport au temps d'exécution. Heureusement, la programmation générique avec la STL [13] permet d'effectuer plusieurs prototypes du même algorithme avec différentes structures de données très rapidement.

Nous venons de constater qu'améliorer la complexité en temps et en espace d'un algorithme passe par le choix de structures de données adaptées aux actions spécifiques de l'algorithme. Cette étape est donc nécessaire mais pas forcément suffisante pour rendre un algorithme échelonnable. Dans l'exemple avec les polygones, une problématique peut apparaître lorsque l'ensemble initial des polygones ne peut être stocké en mémoire interne. En effet, certains polygones adjacents peuvent ne pas être visibles, ce qui peut modifier le résultat final.

Dans la prochaine section, nous listons les techniques génériques existantes pour le passage à l'échelle d'un algorithme.

1.5 Techniques algorithmiques pour le passage à l'échelle

Jusqu'à présent, nous avons supposé que l'ensemble des données en entrée d'un algorithme peut être stocké dans la mémoire interne de l'ordinateur. Comme nous l'avons expliqué en introduction, cette condition n'est plus assurée avec l'arrivée des nouvelles missions d'observation de la Terre. Les algorithmes d'exploitation de données classiques nécessitent d'être adaptés pour être capables de traiter des grands volumes de données.

Trois approches existent dans la littérature pour le passage à l'échelle d'un algorithme. Ces approches, qui seront décrites par la suite, sont listées ci-dessous [14] :

1. Transformer un algorithme par lots [3] en un algorithme de flots de données.

2. La stratégie qui consiste à diviser pour mieux régner.

3. La réduction de la dimension des données en entrée.

Afin d'illustrer comment les algorithmes par lots sont adaptés, nous allons nous baser sur l'algorithme de tri et analyser comment il peut être adapté à l'aide des techniques présentées.
Un algorithme de tri ([11] Partie II Introduction) consiste à trouver à partir d'une séquence de n nombres $\{a_1, a_2, ..., a_n\}$ une permutation de cette séquence vérifiant $a'_1 \leq a'_2 \leq ... \leq a'_n$. De nombreux algorithmes de tri existent et sont répertoriés dans [11].

3. Un algorithme par lots est un algorithme qui nécessite d'avoir la visibilité sur tout l'ensemble des données en entrée pour résoudre le problème.

1.5.1 Algorithme de flots de données

Aussi appelé algorithme au fil de l'eau, un algorithme de flots de données [15] est un algorithme qui reçoit en entrée un flux de données dans le temps. Cet algorithme résout le problème posé pour chaque nouvelle observation sans connaître les observations à venir. Ce type d'algorithme n'a donc pas une vision sur l'ensemble des données qu'il va traiter. L'avantage est qu'il n'y a plus de contrainte mémoire dans la mesure où l'algorithme est capable de résoudre le problème à partir seulement des observations courantes.
Transformer un algorithme par lots en algorithme de flots de données revient à considérer l'ensemble des données en entrée comme un flux de données, où à chaque instant, une donnée est présentée à l'algorithme. Cependant, cette modification nécessite aussi de transformer les étapes et les structures de données utilisées par l'algorithme classique.
Par exemple dans [16], les auteurs proposent un algorithme de tri adapté en algorithme au fil de l'eau. En effet, la séquence de nombres initiale est traversée dans les deux directions (du premier élément vers le dernier élément et du dernier élément vers le premier élément). Dans la première direction, le nombre courant est comparé aux 2 nombres suivants dans la séquence. Dans la seconde direction, le nombre courant est comparé avec les deux nombres qui le précèdent dans la séquence. Cette opération est répétée $N/3$ fois où N est la longueur de la séquence de nombres. Le pseudo-code de l'algorithme est représenté par l'algorithme 1.7.

Pour chacun des nombres $S[l]$ et $S[r]$, seuls $S[l+1]$, $S[l+2]$, $S[r-1]$ et $S[r-2]$ sont nécessaires, la complexité en espace est alors constante $O(1)$. Si n est le nombre de données alors il y a $\frac{n}{3}$ itérations (ligne 4). Pour chaque itération, la boucle **tant que** effectue au maximum n itérations où un nombre constant d'opérations élémentaires sont effectuées. La complexité en temps de l'algorithme est quadratique $O(n^2)$. Il est intéressant de noter que la logique utilisée par l'algorithme est différente de celles présentées dans [11] et qu'elle permet tout de même d'obtenir un résultat identique tout en s'affranchissant de la contrainte mémoire. En revanche, sa complexité en temps, qui est quadratique, peut rendre cet algorithme inutilisable sur des grands volumes de données à cause d'un nombre trop important d'accès en lecture et écriture dans la mémoire externe. Cette technique de passage à l'échelle ne paraît pas appropriée pour l'algorithme de tri.

1.5.2 Diviser pour mieux régner

Cette stratégie [17] consiste à décomposer récursivement l'ensemble des données initiales en blocs de données pouvant être stockés en mémoire interne et à appliquer l'algorithme sur ces blocs. Le résultat final est construit par le regroupement des solutions issues de chaque bloc.
Un exemple d'algorithme de tri utilisant cette stratégie est l'algorithme MergeSort ([11] Partie I Chapitre 2 Section 2.3.1). Dans un premier temps, l'algorithme divise récursivement la séquence de nombres en sous-séquences jusqu'à obtenir des sous-séquences contenant un seul nombre. Ces sous-séquences sont donc forcément triées. Dans un second temps, les sous-séquences sont récursivement fusionnées en maintenant le tri des

```
1:  procédure TriEnLigne(S)
2:      start = 0
3:      N = S.size()
4:      pour i = 0 ; i < N/3 ; i++ exécute
5:          l = start
6:          r = N − 1
7:          tant que l < N − 2 ou r > −1 exécute
8:              si S[l] > S[l + 1] alors
9:                  Échange S[l] et S[l + 1]
10:             fin si
11:             si S[l] > S[l + 2] alors
12:                 Échange S[l] et S[l + 2]
13:             fin si
14:             si S[r] < S[r − 1] alors
15:                 Échange S[r] et S[r − 1]
16:             fin si
17:             si S[r] < S[r − 2] alors
18:                 Échange S[r] et S[r − 2]
19:             fin si
20:             l++
21:             r−−
22:         fin tant que
23:     fin pour
24: fin procédure
```

Algorithme 1.7 – Algorithme de tri adapté en algorithme par flots de données.

nombres dans les séquences résultantes. Le processus s'arrête lorsque nous obtenons une seule séquence triée de tous les nombres. Le pseudo-algorithme est disponible dans [11] Partie I Chapitre 2 Section 2.3.1. Une analyse de la complexité y est faite et vaut $O(n \log n)$. Il s'avère que cette complexité est la meilleure que nous puissions obtenir pour un algorithme de tri lorsque nous comparons les nombres deux à deux [4].
Lorsque la séquence des nombres à trier ne peut être stockée en mémoire interne, une variante de cet algorithme est utilisée. La solution consiste à utiliser la mémoire externe pour stocker des sous-séquences triées de manière temporaire. Une implémentation de cet algorithme est disponible dans la librairie STXXL [18] et se nomme External Merge Sort. Dans un premier temps, la séquence de nombres est divisée en sous-séquences de nombres pouvant être stockées dans la mémoire interne. Avant d'être stockée, chaque

4. En effet, il existe des algorithmes de tri qui n'effectuent pas de comparaisons de nombres. C'est le cas du bin sort, du counting sort et du radix sort. Ces algorithmes ont des complexités en temps linéaires mais des complexités en espace plus importantes.

sous-séquence est triée avec un algorihme de tri classique. L'opération suivante consiste à regrouper plusieurs sous-séquences triées pour former une nouvelle séquence de nombres triée. Soit N le nombre de sous-séquences qui sont fusionnées à chaque étape. N tampons sont alors alloués pour lire les nombres dans les sous-séquences et un tampon est alloué pour écrire la nouvelle séquence de nombres. Ainsi, à chaque étape le nombre minimum est extrait des N valeurs au début de chaque tampon pour être écrit dans le tampon de sortie. Le processus s'arrête lorsqu'il reste qu'une seule séquence. Elle représente la séquence initiale triée. L'ensemble de la procédure est résumé dans la figure 1.7 dans le cas où 2 séquences peuvent être fusionnées à la fois.

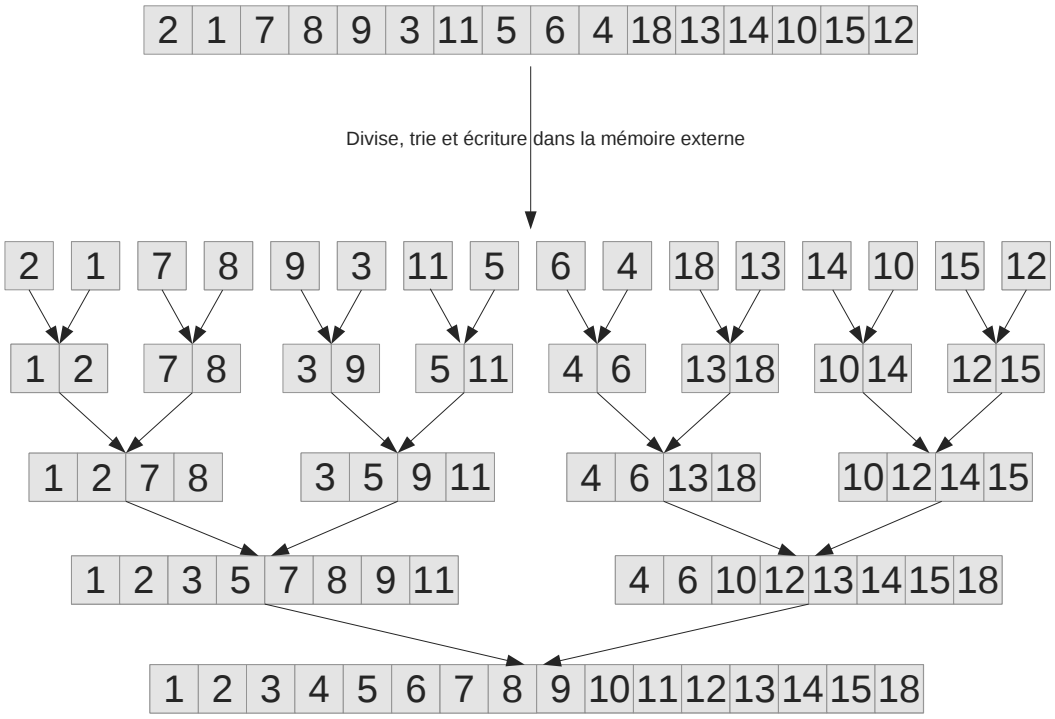


Figure 1.7 – Algorithme External Merge Sort dans la cas où 2 séquences peuvent être fusionnées à la fois.

1.5.3 Réduction de la dimension des données

Cette technique est complètement différente des deux techniques précédentes et ne s'applique pas à l'algorithme de tri, mais seulement aux algorithmes d'apprentissage ou d'estimation. L'objectif de cette technique est d'effectuer un prétraitement sur les données en entrée afin de construire un sous-ensemble représentatif qui peut être contenu en mémoire interne.
Une première solution consiste à effectuer un échantillonnage des données dans l'ensemble

de départ. L'échantillonnage peut-être effectué de manière aléatoire ou périodique.
L'échantillonnage aléatoire consiste à choisir des données de telle sorte que chaque donnée de l'ensemble de départ ait une probabilité égale d'appartenir à l'échantillon. Le tirage peut-être effectué avec ou sans remise. Un avantage de cette méthode est qu'elle est représentative puisque chaque donnée a une chance égale d'appartenir à l'échantillon final. Cependant, cette méthode n'est applicable que lorsque l'ensemble des données initial représente de manière exhaustive l'information.
L'échantillonnage périodique ou encore appelé échantillonnage systématique revient à définir une fréquence de sélection des données. Supposons qu'il y ait N données dans l'ensemble de départ et que chaque donnée soit numérotée de 1 à N. Un exemple d'échantillonnage systématique est de considérer au départ une donnée i et de sélectionner toutes les données d'indice $i + kd$.
Bien que l'échantillonnage soit une technique permettant de réduire la taille des données, il y aura toujours un degré d'incertitude associé aux estimations établies, qui dépend de la méthode choisie mais aussi de la taille de l'échantillon.

Les méthodes de réduction de la dimension des données constituent une piste pour pallier la contrainte de la mémoire.
L'analyse en composantes principales (ACP) [19] permet de réduire le nombre de dimensions d'un jeu de données en cherchant à maximiser la variance tout en décorrélant les nouvelles composantes. L'opération consiste à projeter les données initiales dans un autre repère orthogonal où les vecteurs de la nouvelle base définissent des axes où les données sont mieux distribuées. Techniquement, cela revient à déterminer les vecteurs propres de la matrice de covariance et à sélectionner les d valeurs propres les plus importantes correspondant à la dimension du nouvel espace des données réduites. Dans [20], la méthode d'analyse en composantes indépendantes est utilisée lorsque la décorrélation de l'ACP ne suffit pas à rendre les données indépendantes et cherche à maximiser l'indépendance statistique entre les données.
Le problème des approches par projection est qu'on crée de nouvelles composantes qui sont difficiles à interpréter. De plus, pour calculer les projections, il est nécessaire d'avoir toutes les composantes en mémoire dans un premier temps. Pour pallier ce problème, d'autres techniques consistent à sélectionner seulement les attributs pertinents. La recherche des attributs pertinents est caractérisée par une exploration de l'espace des attributs pour identifier à l'aide d'un critère d'évaluation les attributs pertinents. Dans [21], les auteurs proposent une méthode basée sur le SVM pour sélectionner certains attributs à partir des données hyperspectrales. Dans [22], les auteurs considèrent les attributs sélectionnés pour construire les arbres de décisions comme les attributs pertinents. Sans douter de l'efficacité de telles méthodes, l'inconvénient est qu'elles utilisent parfois des algorithmes complexes pour sélectionner les attributs. De telles méthodes peuvent nécessiter d'avoir la visibilité sur l'ensemble des données, ce qui peut devenir problématique lorsque celui ne peut être stocké en mémoire interne.

1.6 Introduction à la stabilité des algorithmes échelonnables

Le concept de stabilité est utilisé pour les algorithmes numériques [23]. La stabilité se réfère à la propagation des erreurs au cours du traitement, à la capacité de l'algorithme de ne pas trop amplifier d'éventuels écarts et à la précision des résultats obtenus.
Dans la section précédente, nous avons décrit différentes techniques permettant d'adapter un algorithme pour le traitement de données ne pouvant être stockées en mémoire interne. Cette adaptation peut s'effectuer par l'utilisation de structures de données différentes, par le changement du problème à résoudre ou la modification des données en entrée. Toutes ces actions peuvent potentiellement avoir un impact sur la qualité du résultat obtenu. Par analogie à la stabilité numérique, nous définissons le terme de stabilité pour un algorithme échelonnable.
Nous introduisons la notation suivante. Un algorithme qui nécessite d'avoir en mémoire interne l'ensemble des données ainsi que les structures de données utilisées au cours du traitement est noté A_{ref}. L'algorithme noté A_{sca} représente l'algorithme échelonnable construit en adaptant A_{ref} pour le traitement de volumes de données ne pouvant être stockés en mémoire interne. Soit $E = \{e_i\}_{i=1}^{n}$ l'ensemble des données en entrée. L'application d'un algorithme sur E produit un ensemble de données en sortie noté $S_* = \{s_i\}_{i=1}^{m}$ tel que $A_*(E) = S_*$ avec $A_* \in \{A_{ref}, A_{sca}\}$ et $S_* \in \{S_{ref}, S_{sca}\}$. Montrer qu'un algorithme A_{sca} est stable consiste à comparer S_{ref} et S_{sca} et montrer qu'ils sont identiques. Formellement, si nous considérons la fonction identité $I_{ref \to sca}$ de S_{ref} dans S_{sca} :

$$\forall s_i \in S_{ref}, I_{ref \to sca}(s_i) = s_j \text{ avec } s_j \in S_{sca} \text{ et } s_i = s_j \tag{1.1}$$

alors montrer que A_{sca} est stable revient à démontrer que $I_{ref \to sca}$ est bijective, c'est-à-dire que tout élément dans S_{sca} a un et un seul antécédent dans S_{ref}. Pour valider la stabilité de nos algorithmes échelonnables lors des travaux sur la segmentation et la classification, nous montrerons que le cardinal de S_{ref} noté $|S_{ref}|$ est égal à $|S_{sca}|$ et que, pour chaque élément s_i dans S_{ref}, il existe $I_{ref \to sca}(s_i)$ dans S_{sca}.

Segmentation

Chapitre 2

Segmentation par fusion de régions

2.1 Introduction

La segmentation est une procédure qui consiste à partitionner une image I en segments disjoints S_i, avec $i \in [1, \ldots, N]$, selon un critère d'homogénéité :

$$\begin{cases} I = \bigcup_{i=1}^{N} S_i \\ S_i \cap S_j = \emptyset \text{ si } i \neq j \end{cases} \tag{2.1}$$

Un segment représente un ensemble de pixels connectés dans l'image. S_i et S_j sont des segments dits adjacents s'ils partagent une frontière commune. L'image résultante obtenue après une procédure de segmentation est généralement composée de pixels étiquetés où chaque étiquette correspond à un segment. Ainsi, des pixels possèdent la même étiquette s'ils appartiennent au même segment. Dans la suite, nous utilisons le terme objet d'intérêt pour les objets contenus dans la scène (arbre, voiture, culture, *etc*) que nous voulons détecter. Le niveau de détail peut varier suivant l'application. Un utilisateur peut vouloir distinguer les zones forestières, les zones agricoles et les zones urbaines tandis qu'un autre utilisateur peut vouloir distinguer différents types d'arbres dans une même forêt. Par conséquent, il n'existe pas un résultat de segmentation optimal et fréquemment dans une même image segmentée, des zones peuvent être soit sur-segmentées soit sous-segmentées. Nous parlons de sur-segmentation lorsque des objets d'intérêt sont composés de plusieurs segments. À l'inverse, nous parlons de sous-segmentation lorsqu'un objet d'intérêt est combiné avec d'autres objets d'intérêt dans un seul et même segment. Le niveau de détail demandé par l'utilisateur ainsi que l'hétérogénéité des objets d'intérêt contenus dans la scène sont les causes de ces phénomènes. En effet, en reprenant notre exemple, une segmentation visant à distinguer les zones forestières, les zones agricoles et les zones urbaines fournit une sous-segmentation pour celle visant à distinguer les différents types d'arbre dans une forêt.

La formation des segments durant la procédure est basée sur l'utilisation d'un prédicat qui est en général un seuillage sur une fonction de coût calculée à partir d'un critère d'homogénéité :

$$P = \begin{cases} \text{VRAI si coût } < T \\ \text{FAUX sinon.} \end{cases} \tag{2.2}$$

Il existe une multitude de méthodes de segmentation qui peuvent être classées dans différentes catégories (contour, clustering, régions). Plusieurs classifications ont d'ailleurs été proposées dans la littérature [24, 25].

Cette dernière décennie a vu le raffinement de la résolution spatiale de l'imagerie optique. En 1999, le satellite Landsat-7 fournissait des images multispectrales (XS) avec 30 m de résolution spatiale et des images panchromatiques (Pan) avec 10 m de résolution spatiale. Toujours en 1999, Ikonos fournissait des images XS avec 4m et des Pan avec 1m, Quickbird en 2001 (XS : 2.44m, Pan : 0.61m), WorldView-1 en 2007 (Pan : 0.5m), GeoEye en 2008 (XS : 1.65m, Pan : 0.42m), WorldView-2 en 2009 (XS : 1.8m, Pan : 0.46m) et enfin Pléiades en 2012 (XS : 2.8m, Pan : 0.7m). La finesse de la résolution permet d'obtenir une information beaucoup plus détaillée de l'image satellite concernant l'information spectrale, la taille des objets, le contexte spatial et les textures. Cependant, l'exploitation efficace de cette richesse de l'information nécessite des traitements plus complexes. C'est pourquoi, le raffinement de la résolution spatiale a provoqué l'émergence de nouvelles techniques d'analyse en télédétection telles que l'analyse orientée objet [26] (OBIA pour "Object Based Image Analysis") et le raisonnement spatial [27, 28, 29]. Elles utilisent très largement la segmentation [30] pour l'extraction d'objets d'intérêt dans l'image. Ces techniques sont basées sur l'étude d'objets dans la scène plutôt que sur les pixels. En effet, comme expliqué dans [31], les techniques de segmentation basées sur la manipulation des pixels sont devenues obsolètes avec l'arrivée des images à très haute résolution. Dans de telles images, de nombreux artefacts sont présents comme l'ombre des objets et peuvent ainsi affecter le résultat final. En effet, ces artefacts peuvent être considérés comme des objets d'intérêt à part entière. De plus, quand la résolution est très haute, un objet est représenté par des grandes zones contenant de nombreux sous-objets. Les objets d'intéret sont ainsi composites et complexes et ne peuvent plus être décrits efficacement par des indices de texture, des arêtes, *etc.* Il est donc nécessaire d'utiliser des techniques qui permettent un plus haut niveau d'abstraction pour la représentation et la manipulation de l'information contenue dans l'image. C'est pourquoi, nous observons un intérêt et une popularité croissante des approches de segmentation multi-échelle basées sur les régions dans la communauté OBIA.

Dans [32], l'auteur propose 4 types d'approches pour segmenter une image :

- approches basées sur les pixels
- approches basées sur les contours
- approches basées sur les régions
- approches combinant les approches précédentes.

Les approches basées sur les pixels recherchent des zones homogènes dans l'image en se basant sur des seuils sur une ou plusieurs bandes spectrales. Nous pouvons citer l'algorithme du Connected Component [33] ou le Mean-shift [34]. Un inconvénient de ces approches est qu'elles ne considèrent pas les caractéristiques géométriques ainsi que les relations sémantiques des objets dans l'image qui sont apportées par la très haute résolution. De plus, elles sont particulièrement sensibles aux artefacts présents dans l'image.

Les approches basées sur les contours décrivent les objets dans l'image par leur frontière. Nous pouvons citer des algorithmes basés sur l'utilisation du filtre de Canny ou Sobel. Par exemple dans [35], les auteurs détectent d'abord les contours puis remplissent l'intérieur des contours avec des segments en utilisant l'algorithme de ligne de partage des eaux [36]. Le principal inconvénient des approches basées sur les contours est la sensibilité au bruit présent dans l'image qui peut induire une sur-segmentation.

Les approches basées sur les régions manipulent des segments, qui sont constitués d'un ensemble de pixels connectés, pour les fusionner selon un critère d'homogénéité. Ces méthodes sont moins sensibles au bruit dans l'image et permettent d'intégrer de l'information basée sur des représentations à un plus haut niveau d'abstraction comme par exemple l'agencement spatial, la frontière avec les segments voisins, des considérations géométriques, *etc.* Ces approches sont ainsi appropriées pour la segmentation des images à très haute résolution et c'est pour cette raison que nous avons choisi d'étudier leur passage à l'échelle dans cette thèse.

La prochaine section établit une revue des méthodes de segmentation basées sur les régions. Nous allons voir qu'elles englobent elles-même différentes approches pour obtenir une partition de l'image.

2.2 Segmentation basée sur les régions

Les méthodes de segmentation basées sur les régions ne manipulent pas des pixels mais des groupes de pixels (appelés régions ou segments) durant la procédure de segmentation. En général, des opérations, telles que la croissance, la fusion et la division des segments, sont employées pour obtenir une partition de l'image.

Les méthodes basées sur la croissance des régions [37] consistent, à partir d'un ensemble de pixels sélectionnés à l'étape initiale (appelés aussi graines), à faire croître des régions en accumulant des pixels autour de ces graines. L'ajout d'un pixel dans une région est basé sur un critère d'homogénéité. La croissance des régions s'arrête lorsqu'aucun pixel ne peut-être ajouté à aucune des régions. Différentes stratégies existent pour sélectionner les graines. Dans [38], les auteurs effectuent d'abord une détection des contours dans l'image. Les graines sont ensuite choisies entre les contours. Une croissance des régions est effectuée spatialement en associant judicieusement chaque pixel de l'image à une des graines. Dans [39], les auteurs proposent une procédure itérative pour faire croître les régions en utilisant un critère d'homogénéité basé sur la variance des segments. Cette méthode fournit en général une sur-segmentation car elle est sensible

aux grandes variations spectrales dans l'image telles que les zones d'ombre.

Les méthodes basées sur la division de segments [40] considèrent l'image comme un segment initial. Ce segment est divisé récursivement en segments de plus faible taille jusqu'à vérifier un certain critère d'homogénéité. La division peut-être effectuée avec différentes méthodes comme celle des quadtree [41]. L'idée consiste à partitionner récursivement l'image en quadrants de plus en plus petits jusqu'à satisfaire un critère d'homogénéité. Chaque quadrant peut être divisé en 4 quadrants. Cette méthode permet par construction d'obtenir une segmentation hiérarchique de l'image. Pour des images présentant de grandes zones spectrales homogènes, cette méthode donne des résultats corrects. Cependant, un inconvénient des méthodes basées sur la division est qu'elles ont tendance à produire des images sur-segmentées car la division d'un segment produit toujours un nombre fixe de segments de plus faible taille. Par exemple, avec la méthode des quadtree un segment est toujours divisé en 4 segments alors que peut-être une division en 2 ou 3 segments aurait produit une solution plus homogène. C'est pourquoi, dans [42], les auteurs proposent d'effectuer, après une étape de division, une étape de fusion des segments pour fusionner les segments adjacents similaires en utilisant le même critère d'homogénéité utilisé pour l'étape de division.

Enfin, les méthodes basées sur la fusion des régions [43] considérent à l'étape initial chaque pixel de l'image comme un segment. À chaque itération, des paires de segments sont fusionnées pour former un segment plus large. La décision de fusion est basée sur un critère local d'homogénéité qui décrit la similarité des segments adjacents. Cette similarité est mesurée en calculant, pour chaque fusion possible, un coût de fusion entre les paires de segments adjacents. Ces coûts représentent ainsi le degré de similarité. Un seuil T sur le degré de similarité peut être défini soit par l'utilisateur soit par des méthodes automatiques qui cherchent à maximiser la similarité des segments résultants par rapport à une carte de référence. Si le degré de similarité entre deux segments adjacents est inférieur à T, alors les segments adjacents peuvent fusionner. La procédure d'une segmentation par fusion s'arrête lorsqu'il n'y a plus de fusions possibles.

Comparées aux autres approches de segmentation basées sur les régions, les méthodes basées sur la fusion itérative des régions fournissent des segments résultants de bonne qualité dans la mesure où elles s'adaptent à des objets de taille hétérogène dans l'image [44] en particulier avec l'utilisation du critère de Baatz & Schäpe [45] combinant à la fois l'information spectrale et l'information spatiale. La méthode de segmentation par fusion de régions utilisant ce critère est d'ailleurs celle utilisée dans *eCognition* qui, comparée à d'autres méthodes, donne les meilleurs résultats pour le traitement d'images à très haute résolution [32, 46]. C'est pourquoi, nous avons choisi d'étudier le comportement des méthodes de segmentation par fusion de régions et leur passage à l'échelle.

2.3 Différents critères locaux d'homogénéité

Dans cette section, nous allons décrire quelques critères locaux d'homogénéité pour les méthodes de segmentation par fusion de régions.

Un critère local d'homogénéité est utilisé pour calculer une mesure de similarité entre les segments adjacents d'une image. Si cette mesure de similarité est inférieure à un seuil, généralement défini par l'utilisateur, alors les segments sont des candidats potentiels pour fusionner.
Il existe plusieurs familles de critères locaux d'homogénéité. La première distinction entre ces familles se base sur le type d'information pour calculer la mesure de similarité. L'information peut être spectrale, spatiale, spectrale et spatiale, *etc.* Certains critères peuvent prendre en compte l'homogénéité du segment qui résulterait de la fusion des segments adjacents pour calculer la mesure de similarité. Dans la suite, nous listons quelques exemples de critères locaux d'homogénéité et nous identifions leurs caractéristiques spécifiques pour calculer la mesure de similarité.

Dans [47], les auteurs proposent un critère d'homogénéité basé sur la distance de Fisher. Soit S_1 et S_2, une paire de segments adjacents où a_i et σ_i sont respectivement la surface et l'écart-type du segment S_i. La surface est le nombre de pixels contenus dans le segment. L'écart-type est celui des intensités spectrales des pixels contenus dans S_i pour chaque bande. σ_i est donc la somme des écart-types pour chaque bande spectrale. Si le prédicat suivant est vérifié :

$$\frac{(a_1 + a_2) \times (\sigma_1 + \sigma_2)^2}{a_1\sigma_1^2 + a_2\sigma_2^2} < T \tag{2.3}$$

alors S_1 et S_2 peuvent fusionner. Ce critère est donc basé seulement sur l'information spectrale et prend en compte l'écart-type résultant du segment issu de la fusion de S_1 et S_2.

Dans [48], les auteurs proposent un critère basé seulement sur la distance euclidienne entre les vecteurs des moyennes spectrales notés μ_1 et μ_2 pour une paire de segments adjacents S_1 et S_2. Dans leur implémentation, les auteurs proposent d'intégrer un seuil $T(t)$ qui s'incrémente au fil des itérations. Par conséquent, les fusions initiales sont plus difficiles à accomplir que celles à la fin de la procédure. En notant T_{lim} le seuil final, le prédicat à vérifier est le suivant :

$$||\mu_1 - \mu_2|| < T(t) \text{ avec } T(t) = 1, 2 \ldots, T_{lim} \tag{2.4}$$

Ce critère a été intégré dans le logiciel de traitement d'image SPRING [49] et récemment dans le logiciel libre InterImage [50]. Il est basé seulement sur l'information spectrale et ne prend pas en compte l'évaluation de l'homogénéité du segment résultant de la fusion.

Le raffinement de la résolution spatiale a soulevé un intérêt pour la création de critères d'homogénéité combinant l'information spectrale et l'information spatiale.
Dans [45], les auteurs proposent de combiner ces deux informations pour proposer un critère capable de s'adapter à des images de résolutions différentes et à des objets d'intérêt de tailles différentes. Ce critère est nommé le critère de Baatz & Schäpe[45]. Chaque segment est caractérisé par des attributs spectraux et géométriques. L'attribut spectral, noté $\sigma_i[b]$, est l'écart-type des intensités spectrales des pixels contenus dans le segment pour la bande spectrale b. Les attributs géométriques sont la surface a_i de S_i, le périmètre p_i de S_i et la boîte englobante bb_i qui représente le rectangle de taille minimum

contenant S_i dont les côtés sont parallèles aux axes de l'image. Le coût de fusion entre deux segments adjacents S_i et S_j, noté $f_{i,j}$, représente l'augmentation de l'hétérogénéité à la fois spectrale et spatiale au sein du segment résultant. Le seuil, pour déterminer si S_i et S_j peuvent fusionner, est le facteur d'échelle, noté s, qui influence directement la taille des segments résultants. Ainsi, S_i et S_j peuvent fusionner si $f_{i,j} < s^2$.
L'augmentation de l'hétérogénéité spectrale, notée f_c, est basée sur la différence des écart-types entre le segment résultant $S_{i,j}$ et les segments fusionnés S_i et S_j :

$$f_c = \sum_b \Big[(a_i + a_j) \times \sigma_{i,j}[b] - (a_i \times \sigma_i[b] + a_j \times \sigma_j[b])\Big] \tag{2.5}$$

L'augmentation de l'hétérogénéité spatiale, notée f_s, est basée sur le degré de lissage et de compacité du segment résultant $S_{i,j}$ par rapport à S_i et S_j. Le degré de lissage est défini par le ratio entre le produit $a_i \times p_i$ et le périmètre de bb_i. Le degré de compacité est défini par le produit entre $\sqrt{a_i}$ et le périmètre p_i. Avec ces notations, l'augmentation de l'hétérogénéité du degré de lissage d_l s'exprime de la façon suivante :

$$d_l = \frac{a_{i,j} \times p_{i,j}}{len(bb_{i,j})} - \left(\frac{a_i \times p_i}{len(bb_i)} + \frac{a_j \times p_j}{len(bb_j)}\right) \tag{2.6}$$

et l'augmentation de l'hétérogénéité du degré de compacité d_c s'exprime :

$$d_c = \sqrt{a_{i,j}} \times p_{i,j} - (\sqrt{a_i} \times p_i + \sqrt{a_j} \times p_j) \tag{2.7}$$

Ainsi, f_s s'exprime :

$$f_s = w_s \times d_c + (1 - w_s) \times d_l \tag{2.8}$$

avec $w_s \in [0, 1]$ indiquant l'importance relative de d_c par rapport à d_l.
Finalement, l'expression de l'augmentation de l'hétérogénéité globale f est la suivante :

$$f = w_c \times f_c + (1 - w_c) \times f_s \tag{2.9}$$

avec $w_c \in [0, 1]$ indiquant l'importance relative de f_c par rapport à f_s.
Ainsi, ce critère nécessite la configuration de 3 paramètres s, w_c et w_s pour éviter la sur-segmentation et la sous-segmentation. Ce critère est celui utilisé dans le logiciel de traitement d'images propriétaire *eCognition* et donne des résultats de très bonne qualité [51]. Ce critère combine l'information spatiale et l'information spectrale. De plus, il évalue l'homogénéité du segment résultant issu de la fusion.

Dans [52], les auteurs proposent un critère combinant l'information spectrale et l'information spatiale nommé Full Lambda Schedule. Le coût de fusion s'exprime de la façon suivante :

$$\frac{\frac{a_i \times a_j}{a_i + a_j} \times ||\mu_i - \mu_j||^2}{len(\partial(S_i, S_j))} < T \tag{2.10}$$

avec $len(\partial(S_i, S_j))$ la longueur de la frontière partagée par S_i et S_j. Par rapport au critère BS, ce critère ne tient pas compte de l'hétérogénéité du segment résultant mais

se base seulement sur la similarité entre S_i et S_j. Une valeur importante de T induit des segments avec des surfaces importantes et des frontières communes avec leurs segments adjacents faibles.

De nombreux autres critères existent utilisant d'autres types d'attributs (statistiques, temporels, *etc*) et sont décrits dans [53, 54, 55, 56, 57, 58].

2.4 Différentes heuristiques pour la fusion des segments

Nous venons de lister dans la section précédente des exemples de critères d'homogénéité utilisés pour fusionner les segments. La procédure de segmentation par fusion de régions consiste à calculer, à chaque itération, des coûts de fusion entre les segments adjacents. Nous avons vu que la première condition pour qu'une paire de segments soit fusionnée est qu'elle respecte un prédicat, c'est-à-dire que le coût de fusion soit inférieur à un seuil T défini généralement par l'utilisateur.
Cette condition est nécessaire mais elle n'est pas suffisante, car, pour un segment donné, il peut exister plusieurs segments adjacents pour lesquels le prédicat est vérifié. Différentes heuristiques pour la sélection d'un segment adjacent ont été proposées et sont répertoriées dans [45].

Fitting (F)

Étant donné un segment S_1 et la liste de ses segments adjacents, cette heuristique consiste à choisir le premier segment adjacent S_2 pour lequel le prédicat est vérifié avec S_1.

Best Fitting (BF)

Étant donné un segment S_1 et la liste de ses segments adjacents, cette heuristique consiste à choisir un segment adjacent S_2 pour lequel le coût de fusion avec S_1 est le minimum parmi ceux qui vérifient le prédicat.

Local Mutual Best Fitting (LMBF)

Soit un segment S_1 et la liste de ses segments adjacents. Soit S_2 le segment adjacent pour lequel le coût est le minimum parmi ceux qui vérifient le prédicat. Dans cette heuristique, une contrainte supplémentaire est ajoutée et consiste à déterminer le segment adjacent de S_2 pour lequel le coût est minimum parmi ceux qui vérifient le prédicat. Nous notons ce segment S_3. Si $S_3 = S_1$ alors les segments S_1 et S_2 sont dîts mutuellement similaires et sont fusionnés.

Global Mutual Best Fitting (GMBF)

Contrairement aux autres heuristiques, cette heuristique implique de ne fusionner qu'une paire de segments adjacents à chaque itération. La paire de segments adjacents qui fusionne est celle pour laquelle la mesure de similarité est la minimum parmi celles de toutes les autres paires de segments adjacents de l'image. Cette heuristique est la plus contraignante car seulement une paire de segments fusionne à chaque itération, ce qui peut augmenter sévèrement le temps d'exécution de l'algorithme.

Ordre de traitement des segments

Toujours dans [45], les auteurs expliquent l'importance de faire fusionner les segments dans l'image de façon simultanée pour garantir que leurs tailles soient globalement homogènes. À l'exception de l'heuristique GMBF, un même ordre de visite des segments peut entraîner un déséquilibre au niveau de leurs tailles. En effet, ce sont toujours les mêmes segments qui vont potentiellement fusionner en premier. La solution proposée est l'utilisation d'une matrice de réarrangement. Elle a pour but de visiter tous les segments de l'image tout en imposant que 2 fusions de paires de segments successives soient éloignées spatialement. Pour ce faire, les auteurs utilisent une matrice dont le nombre de cases est supérieur ou égal au nombre de segments et où chaque case contient la position d'un segment. Cette matrice est construite de telle manière que 2 cases adjacentes contiennent les positions de segments éloignés spatialement dans l'image. Un exemple d'une telle matrice de taille 2×2 est représenté ci-dessous :

$$\begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}$$

Les cases de la matrice sont alors exécutées dans l'ordre $(0,0) \rightarrow (1,1) \rightarrow (0,1) \rightarrow (1,0)$.

Une solution, pour rendre le résultat de segmentation indépendant de l'ordre de traitement des segments en utilisant la LMBF, est d'imposer que chaque segment ne puisse fusionner au maximum qu'une seule fois à chaque itération. Cependant, cette contrainte a l'inconvénient d'induire plus d'itérations pour terminer la segmentation. Pour le démontrer, nous considérons 3 segments S_1, S_2 et S_3, où S_2 est le segment adjacent le plus similaire à S_1, S_3 est le segment adjacent le plus similaire à S_2 et S_2 le segment adjacent le plus similaire à S_3. Si nous utilisons l'heuristique BF et que nous visitons les segments dans cet ordre $S_1 \rightarrow S_2 \rightarrow S_3$, alors S_1 et S_2 vont fusionner mais pas S_3 puisque S_2 a déjà fusionné. Par contre, si les segments sont visités dans cet ordre $S_2 \rightarrow S_3 \rightarrow S_1$, alors S_2 et S_3 vont fusionner mais pas S_1. Avec l'heuristique BF, deux ordres de visite différents peuvent induire des segments résultants différents. En revanche, si nous considérons l'heuristique LMBF, les deux ordres de visite précédents produisent les mêmes segments résultants. En effet, nous avons toujours S_2 et S_3 qui fusionnent mais pas S_1. Ce résultat est vrai pour les 6 ordres de visites possibles et de manière générale, pour n'importe quel ordre de visite. Ceci s'explique par le comportement bijectif de l'heuristique LMBF car une paire de segments fusionne seulement si les segments sont mutuellement similaires.

Cela évite la possibilité d'une fusion de l'un des deux segments avec un autre segment adjacent. Par la suite, nous avons sélectionné l'heuristique LMBF avec la contrainte d'imposer une seule fusion au maximum par segment à chaque itération. En effet, cette stratégie permet d'obtenir des résultats stables dans le sens où une permutation des données en entrée n'a aucun effet sur le résultat final.

2.5 Motivations pour l'unification des méthodes par fusion de régions

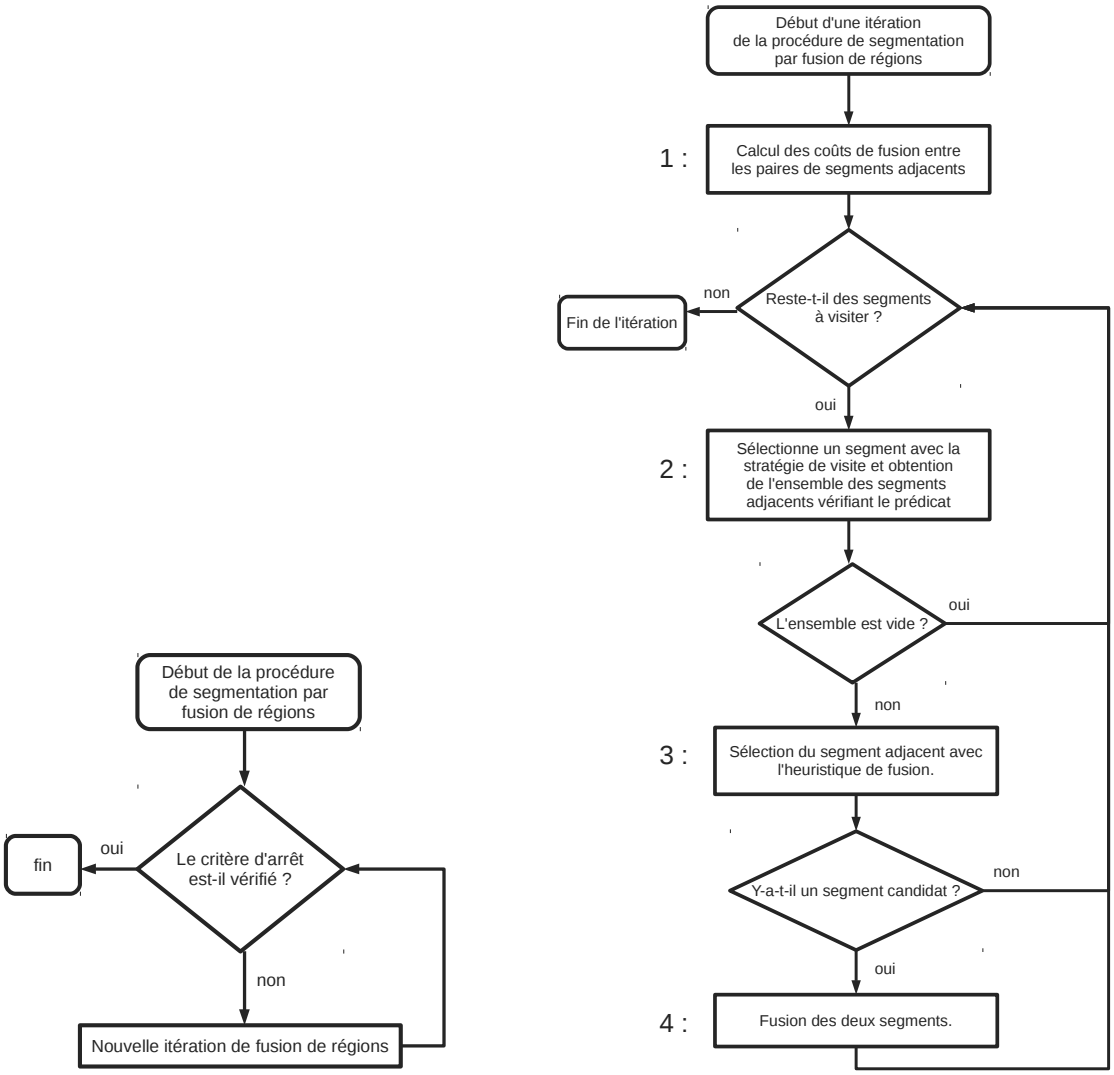
À partir de la description des méthodes par fusion de régions faite dans les sections précédentes, nous avons identifié de nombreuses opérations communes réalisées lors de la segmentation. Un algorithme de segmentation par fusion de régions est décrit par des étapes génériques qui consistent à fusionner les segments au fil des itérations. Un prédicat basé sur un critère d'homogénéité est toujours utilisé pour déterminer les paires de segments adjacents qui peuvent fusionner. Parmi les paires de segments candidates pour la fusion, une heuristique choisit celles qui vont réellement fusionner en se basant sur un critère de sélection. Enfin, un critère d'arrêt, composé d'une ou plusieurs conditions, permet de décider si oui ou non la procédure de fusion de régions doit continuer après chaque itération.
L'ensemble des étapes génériques d'une procédure de segmentation par fusion de régions est représenté dans les figures 2.1a et 2.1b. La figure 2.1a décrit la boucle principale de la procédure qui exécute des nouvelles itérations de fusion de régions tant que le critère d'arrêt global n'est pas vérifié. La figure 2.1b décrit les différentes étapes d'une itération de fusion de régions.

Dans le prochain chapitre, nous décrivons l'algorithme GRM ("Generic Region Merging") développé dans les travaux de cette thèse, qui vise à unifier les méthodes par fusion de régions. L'algorithme GRM est indépendant du critère local d'homogénéité et propose l'utilisation des heuristiques de fusion introduites dans la section précédente. L'utilisateur peut configurer le choix de visite des segments à chaque itération. La première stratégie correspond à l'utilisation d'une matrice de réarrangement qui permet une croissance équilibrée des segments au fil des itérations. Avec l'utilisation de cette stratégie de visite, un segment peut fusionner plusieurs fois au cours d'une itération. La seconde stratégie utilise l'heuristique LMBF et impose que chaque segment ne fusionne qu'une seule fois au maximum à chaque itération. Les segments sont alors visités toujours dans le même ordre.

En revanche, la manière de calculer les coûts et de mettre à jour les attributs spécifiques des segments doit être indiquée par l'utilisateur lors de l'ajout d'un nouveau critère d'homogénéité dans l'algorithme GRM.

Nous informons le lecteur que l'algorithme GRM est déjà intégré dans OTB en tant que Remote Module [1] et propose déjà les critères suivants : Baatz & Schäpe, Full Lambda

1. https://www.orfeo-toolbox.org/external-projects/



(a) Boucle principale **(b)** Une itération

Figure 2.1 – Ensemble des étapes génériques lors de la procédure de segmentation par fusion de régions.

Schedule et le critère basé sur la distance euclidienne entre les moyennes spectrales.

En se basant sur la figure 2.1, le prochain chapitre décrit les structures de données utilisées et les différentes étapes de l'algorithme GRM, ainsi que ses complexités en temps et en espace.

Chapitre 3

Algorithme générique de segmentation par fusion de régions

3.1 Introduction

Au cours du chapitre précédent, les méthodes de segmentation par fusion de régions ont été sélectionnées afin d'étudier leur passage à l'échelle. Ce chapitre a pour objectif d'introduire l'algorithme GRM, qui unifie l'ensemble des méthodes par fusion de régions en proposant un cadre indépendant du critère d'homogénéité et de l'heuristique de fusion. Nous rappelons que GRM est l'acronyme de "Generic Region Merging" pouvant être traduit par fusion générique des régions. En programmation, la généricité consiste à définir des algorithmes identiques opérant sur des données de différents types. Partant du constat qu'il existe de nombreux critères d'homogénéité pour les méthodes de segmentation par fusion de régions, et tirant parti du fait que de nombreuses étapes algorithmiques sont communes pour toutes ces méthodes, nous avons choisi de développer un algorithme générique de fusion de régions acceptant n'importe quel critère local d'homogénéité et n'importe quelle heuristique de fusion. Son pseudo-algorithme est illustré par l'algorithme 3.1.

En effet, plusieurs opérations sur le graphe de segments sont génériques :

- L'initialisation des attributs internes des segments et leur voisinage (ligne 2).
- Le calcul des coûts de fusion entre les différentes paires de segments adjacents (ligne 4).
- L'heuristique permettant de sélectionner parmi les paires de segments qui vérifient le prédicat celles qui vont fusionner (ligne 9).
- La fusion des segments qui induit la formation d'un nouveau segment à partir

```
1:  procédure SEGMENTATION(image)
2:      Initialisation des segments.
3:      tant que le critère d'arrêt n'est pas vérifié exécute
4:          Calcul des coûts de fusion entre les segments adjacents
5:          tant que il reste des segments à visiter exécute
6:              Sélectionne un segment S_i avec la stratégie de visite.
7:              Établit l'ensemble des segments adjacents vérifiant le prédicat.
8:              si l'ensemble n'est pas vide alors
9:                  Sélectionne un segment S_j avec l'heuristique de fusion.
10:                 si S_j existe alors
11:                     Fusion de S_i et S_j.
12:                 fin si
13:             fin si
14:         fin tant que
15:         Suppression des segments fusionnés.
16:     fin tant que
17: fin procédure
```

Algorithme 3.1 – Procédure de la segmentation par fusion de régions

d'une paire de segments adjacents ainsi que la mise à jour de son voisinage et de ses attributs (ligne 11).

- La suppression des segments qui ont fusionné (ligne 15).

Par la suite, chacune de ces étapes va être décrite et leurs complexités étudiées. En fonction de la nature des opérations sur les données, un choix judicieux des structures de données sera justifié pour réduire significativement la complexité en espace de l'algorithme pour le rendre plus efficace.

3.2 Structures de données

3.2.1 Représentation basée sur un graphe

Dans l'algorithme GRM, un graphe est utilisé pour représenter les segments dans l'image durant la procédure de segmentation. Chaque noeud du graphe est un segment de l'image et chaque arête représente un lien d'adjacence entre 2 segments. Deux segments sont adjacents s'ils partagent une frontière commune (arêtes de pixels communes).
Au début de la procédure de segmentation, chaque pixel de l'image est un segment et chaque segment initial contient 4 ou 8 segments adjacents suivant le choix de la connexité des pixels. Dans la suite, sans perte de généralité, nous supposons que la connexité est 4. Un segment a donc 4 segments adjacents situés en haut, à droite, en bas et à gauche.

Pour le choix de la structure de données à utiliser pour représenter les segments, il est nécessaire de déterminer au préalable la densité du graphe qui représente "à quel point tout le monde est lié". Ainsi, le graphe initial des segments est composé de noeuds contenant au plus 4 arêtes. En notant $|E|$ le nombre d'arêtes et $|V|$ le nombre de noeuds, la densité du graphe initial peut s'exprimer de la façon suivante :

$$D = \frac{|E|}{|V| \times (|V| - 1)} \tag{3.1}$$

où $|E| = 4 \times |V|$ puisque chaque segment initial contient 4 segments adjacents. Après simplification, la densité initiale du graphe vaut :

$$D = \frac{4}{|V| - 1} \tag{3.2}$$

Lorsque $|V| \to \infty$ alors $D \to 0$. Le graphe des segments est un graphe creux. Pour vérifier que le graphe garde cette propriété durant toute la procédure de segmentation, nous avons segmenté une image Quickbird (figure 3.1a) à très haute résolution sur la ville de Paris de taille 1000×1000 pixels avec l'algorithme GRM et l'heuristique de fusion LMBF. La visite des segments à chaque itération s'effectue toujours dans le même ordre car nous choisissons que chaque segment ne fusionne qu'une fois à chaque itération. Le critère d'homogénéité utilisé est celui de Baatz & Schäpe avec les paramètres suivants : $w_c = 0,7$, $w_s = 0,3$ et $s = 60$. Le critère d'arrêt est vérifié lorsqu'il n'y a plus de fusions possibles. Après chaque itération, nous avons mesuré la densité du graphe de segments. Les densités en fonction du nombre d'itérations sont représentées par le graphe de la figure 3.1c.

La densité du graphe augmente au fil des itérations mais reste tout de même dans des valeurs proches de 0. En effet, la densité initiale du graphe est de l'ordre de 10^{-6} et la densité finale de l'ordre de 10^{-3}. Nous pouvons raisonnablement supposer que le graphe reste creux tout au long de la procédure de segmentation. Utiliser une matrice d'adjacence pour stocker les arêtes entre les noeuds serait une mauvaise stratégie car la complexité en espace engendrée par une telle structure est $O(N^2)$ où N est le nombre de segments. Puisque chaque segment contient peu d'arêtes, comparé au nombre de segments dans l'image, nous avons choisi d'utiliser une liste d'adjacence pour stocker les segments en mémoire interne car elle permet d'utiliser juste la quantité de mémoire nécessaire pour stocker les noeuds et les arêtes. Sa représentation est illustrée par la figure 3.2. Utiliser une telle structure de données a l'avantage d'avoir une complexité en espace linéaire $O(N + E)$ où E est le nombre d'arêtes.

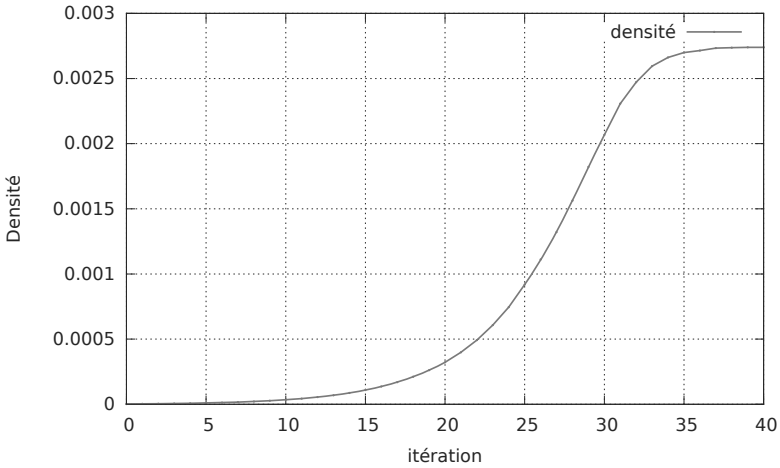
Un graphe est ainsi une liste de noeuds représentant chaque segment de l'image et chaque noeud contient une liste de pointeurs vers les noeuds adjacents.

3.2.2 Représentation d'un segment

L'algorithme GRM fournit une représentation générique d'un segment dans l'image. Lorsque l'utilisateur veut intégrer un nouveau critère d'homogénéité, des attributs spécifiques peuvent être nécessaires et doivent être rattachés à la représentation générique

(a) Image originale

(b) Image segmentée



(c) Densité en fonction du nombre d'itérations

Figure 3.1 – Évolution de la densité du graphe des segments en fonction du nombre d'itérations.

d'un segment.
Un segment générique contient plusieurs informations. Un identifiant unique noté *id* est associé à chaque segment dans l'image. Cet identifiant correspond aux coordonnées du pixel représentant le segment initial et est attribué durant la phase d'initialisation.
Chaque segment contient 3 booléens. Le premier indique si le segment a fusionné lors de l'itération précédente. Ce booléen permet d'éviter de recalculer inutilement des coûts de fusion entre des paires de segments adjacents qui n'ont pas fusionné à l'itération précédente. Le second indique si le segment a fusionné durant l'itération courante. Ce booléen est nécessaire pour éviter de fusionner un segment avec un segment adjacent qui a déjà

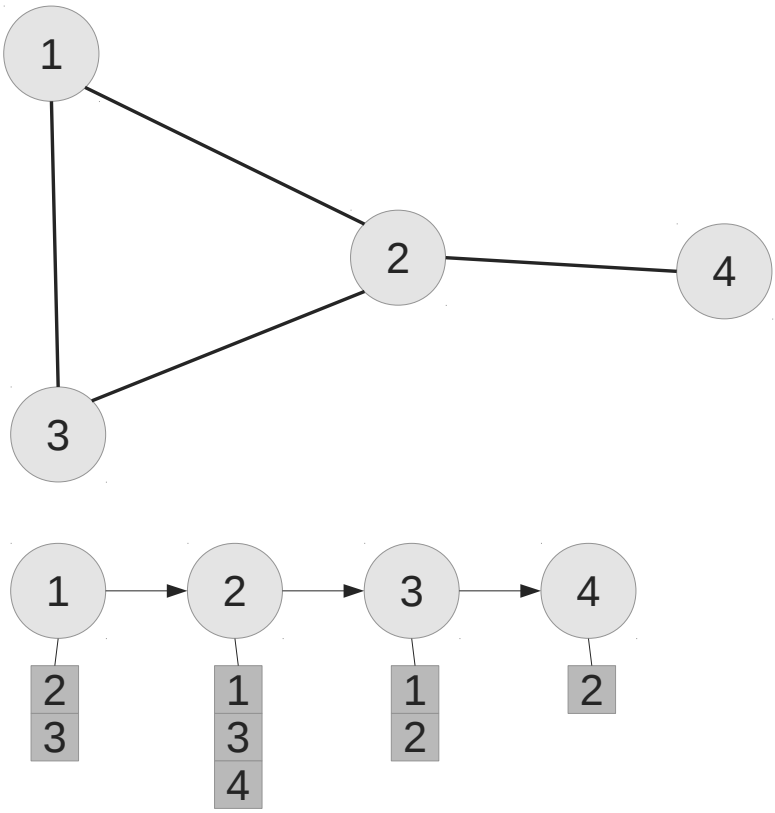


Figure 3.2 – Représentation du graphe des segments avec une liste d'adjacence

fusionné. Enfin, le dernier booléen indique si le segment doit être supprimé du graphe à la fin d'une itération. En effet, un segment qui a fusionné dans un autre segment ne doit plus être considéré dans le graphe.
Chaque segment contient un attribut géométrique qui représente le rectangle minimum englobant le segment et dont les côtés sont parallèles à ceux de l'image. Cet attribut, noté *bbox*, contient la localisation et la dimension du rectangle. Il est illustré par la figure 3.3.
Chaque segment contient la liste des arêtes pointant vers ses segments adjacents notée *aretes*. La représentation d'une arête est décrite dans la section 3.2.3.
Enfin, chaque segment contient également la localisation de tous les pixels contenus dans son contour et est noté *contour*. La structure de données utilisée pour représenter cet attribut est décrite dans la section 3.3.3.

3.2.3 Représentation d'un lien d'adjacence

L'algorithme GRM fournit une représentation générique d'un lien d'adjacence entre 2 segments S_i et S_j. S_i possède une arête notée $e_{i,j}$ vers S_j et S_j possède une arête

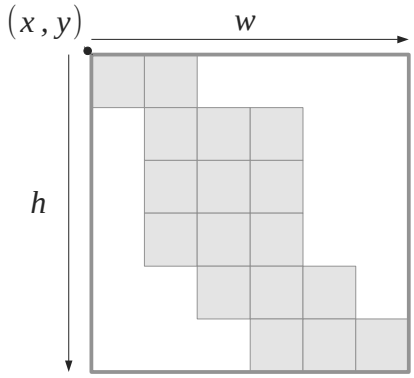


Figure 3.3 – Représentation du rectangle englobant un segment ($bbox$). Cet attribut est composé des coordonnées du coin supérieur gauche du rectangle, notées (x, y), ainsi que de la largeur et la hauteur du rectangle, notées respectivement w et h.

$e_{j,i}$ vers S_i. Chaque arête contient un pointeur vers le segment adjacent, mais aussi le coût de fusion entre les 2 segments noté $c_{i,j}$, ainsi qu'un booléen indiquant si le coût est à jour. Enfin, une arête contient aussi la longueur de la bordure commune entre les segments adjacents, notée $frontiere$, qui est illustrée par la figure 3.4.

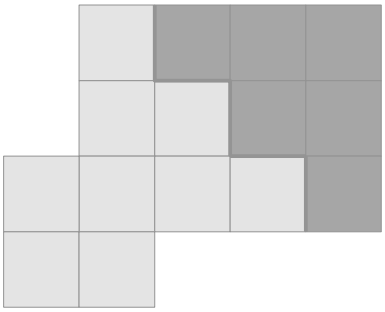


Figure 3.4 – Bordure commune entre 2 segments adjacents. L'arête contient la longueur de cette bordure qui correspond au nombre d'arêtes de pixels communes. Dans cet exemple, $frontiere = 5$.

3.2.4 Représentation du contour d'un segment

Afin de pouvoir localiser chaque segment dans l'image, il est nécessaire de maintenir, tout au long de la procédure de segmentation, les coordonnées des pixels contenus dans chaque segment. Pour des images de petite taille, stocker la liste des pixels internes de chaque segment est la solution évidente. Cependant, pour des images ne pouvant être stockées en mémoire interne, cette stratégie est problématique puisqu'il peut devenir impossible de stocker en mémoire toutes les coordonnées des pixels. En effet, les

coordonnées de chaque pixel sont codées avec le type *long unsigned int*, qui nécessite 8 octets dans la mémoire des processeurs actuels. Par exemple, pour une image Pléiades de 40000×40000 pixels, cela nécessite 12 gigaoctets en mémoire interne juste pour stocker les coordonnées. Il est donc nécessaire de trouver une autre stratégie pour localiser les segments dans l'image. Une première idée fut de considérer seulement les coordonnées des pixels localisés sur les contours des segments. Bien que cette stratégie permette de réduire la quantité de mémoire nécessaire, la réduction peut ne pas être suffisante, surtout lors du traitement d'images satellite à très haute résolution contenant beaucoup d'objets d'intérêt de petite taille.
La stratégie choisie consiste à coder les déplacements le long des contours des segments sous la forme d'une chaîne de Freeman [59]. L'avantage de cette représentation est que chaque déplacement peut-être codé en utilisant seulement 2 bits en mémoire interne. La figure 3.5 illustre les 4 déplacements possibles le long du contour d'un segment.

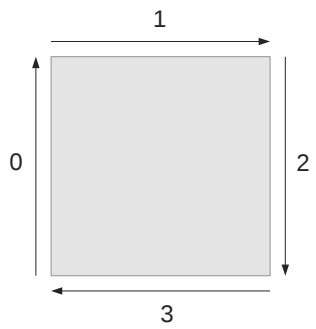


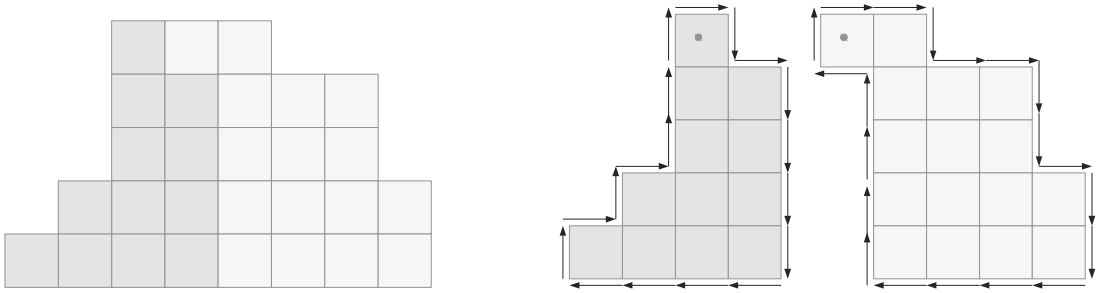
Figure 3.5 – Déplacements possibles le long d'un contour : vers le haut (0), vers la droite (1), vers le bas (2) et vers la gauche (3).

Dans la suite, deux opérations sur les contours, qui sont nécessaires pour la procédure de segmentation, vont être décrites. La première consiste à créer le contour du segment résultant de la fusion de 2 segments adjacents. La seconde opération consiste à générer les coordonnées des pixels internes d'un segment connaissant son contour et sa boîte englobante pour générer l'image étiquetée.

Fusion des contours

Soit une paire de segments S_1 et S_2 qui doivent fusionner. Leurs contours sont illustrés par la figure 3.6. Un contour est représenté par une liste de déplacements le long des pixels situés sur le contour du segment et par les coordonnées du pixel de départ. Ces pixels, représentant le point de départ du contour pour chaque segment, sont marqués par un cercle rouge dans la figure 3.6b. Les identifiants *id* de chaque segment représentent les coordonnées de ces pixels.

En analysant les déplacements le long d'un contour, des combinaisons récurrentes de paires de déplacements successifs peuvent être extraites et sont répertoriées dans la figure 3.7.



(a) S_1(bleu) et S_2 (jaune). **(b)** Contours de S_1 et S_2.

Figure 3.6 – Représentation des contours de S_1 et S_2 qui doivent être fusionnés.

La procédure de fusion de 2 contours est décrite par la suite en se basant sur la fusion des segments illustrés dans la figure 3.6. Le segment issu de la fusion de S_1 et S_2 est noté S_{12}.
La première étape consiste à parcourir les 2 contours de S_1 et S_2 pour générer les coordonnées des pixels situés sur le contour des segments que nous stockons dans une liste notée L_b. La génération des coordonnées des pixels est effectuée grâce à la liste des combinaisons des paires de déplacements répertoriées dans la figure 3.7.
Le contour de S_{12} est ensuite construit en utilisant L_b et la liste des combinaisons de déplacements possibles. Le pixel de départ est celui dont les coordonnées sont les plus proches de l'origine de l'image. Dans l'exemple, le pixel de départ est celui contenu dans S_1 marqué par un cercle rouge (figure 3.6b).
Le contour de S_{12} est représenté par la figure 3.8.

Génération des pixels internes

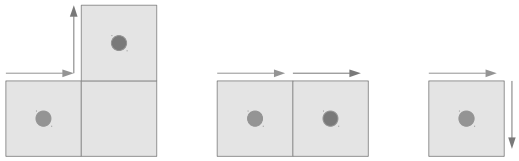
À la fin d'une procédure de segmentation, il est nécessaire de fournir une image étiquetée où tous les pixels appartenant au même segment se voient assigner la même étiquette. La génération de l'image étiquetée s'effectue en 2 étapes.

La première étape consiste à écrire les étiquettes des pixels situés sur les contours des segments. Pour effectuer cela, il est nécessaire de parcourir l'ensemble des segments du graphe, de générer pour chaque segment la liste des coordonnées des pixels situés sur son contour et d'écrire dans l'image pour chaque coordonnée la valeur de l'étiquette du segment. Les pixels, qui n'appartiennent à aucun contour, se voient assigner une étiquette égale à 0.

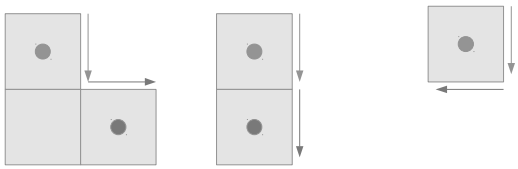
La seconde étape consiste à parcourir l'image de gauche vers la droite et de haut en bas pour étiqueter les pixels internes aux segments. L'étiquette courante est maintenue en mémoire tout au long de la visite. Pour chaque pixel rencontré, il y a 2 cas possibles :

- Soit le pixel contient une étiquette. Dans ce cas, l'étiquette courante devient l'étiquette de ce pixel.

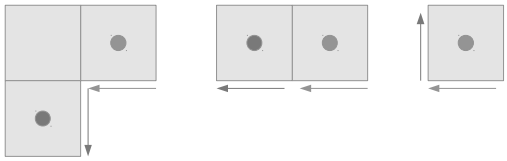
Combinaisons possibles lorsque le premier déplacement est vers **la droite (1) :**



Combinaisons possibles lorsque le premier déplacement est vers **le bas (2) :**



Combinaisons possibles lorsque le premier déplacement est vers **la gauche (3) :**



Combinaisons possibles lorsque le premier déplacement est vers **le haut (0) :**

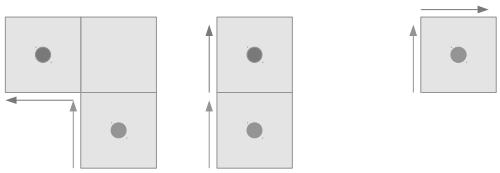


Figure 3.7 – Ensemble des combinaisons de paires de déplacements successifs possibles. Le premier déplacement est indiqué avec une flèche rouge et se déplace le long du pixel marqué avec un cercle rouge. Le second déplacement est indiqué avec une flèche bleue et se déplace le long du pixel marqué avec un cercle bleu.

- Soit le pixel a une étiquette nulle. Dans ce cas, son étiquette a pour valeur l'étiquette courante.

La figure 3.9 illustre l'enchaînement des opérations. Un extrait d'une image fournie par le satellite Ikonos (figure 3.9a) a été segmentée avec l'algorithme GRM en utilisant le critère de Baatz & Schäpe. L'image contenant seulement les étiquettes des pixels situés sur les contours est illustrée par la figure 3.9b. Pour une meilleure visualisation, tous les contours sont noirs mais en réalité chaque contour de chaque segment a une étiquette unique. La figure 3.9c représente l'image étiquetée.

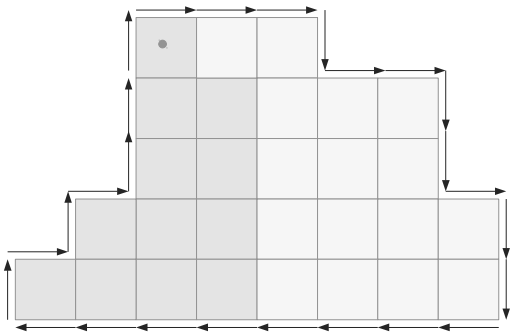
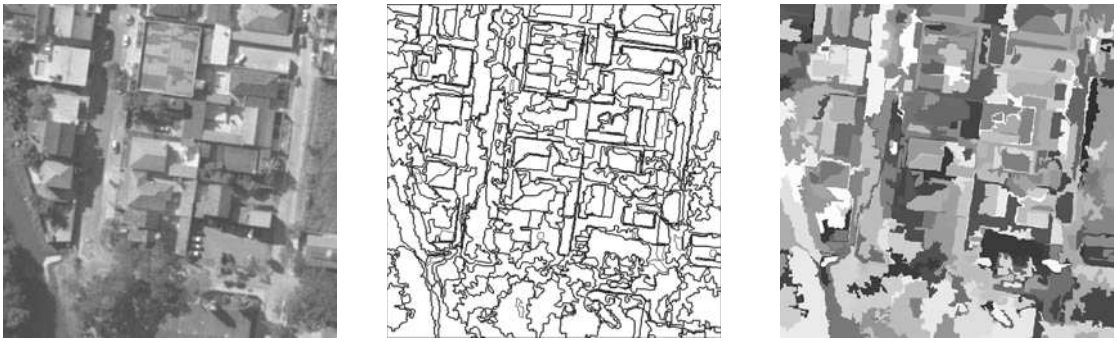


Figure 3.8 – Contour du segment S_{12}.



(a) Image originale **(b)** Image intermédiaire **(c)** Image étiquetée

Figure 3.9 – Génération de l'image étiquetée à partir du contour des segments. L'opération s'effectue en 2 étapes : la première consiste à générer les coordonnées des pixels situés sur les contours des segments et à écrire les étiquettes dans l'image (figure 3.9b). La seconde étape consiste à écrire les étiquettes des pixels à l'intérieur des segments(figure 3.9c).

Réduction de la mémoire

Le déplacement d'un contour peut-être codé sur 2 bits. La structure de données utilisée dans l'algorithme GRM pour stocker les déplacements le long d'un contour est une table dynamique manipulant des bits. Il existe plusieurs types de tables dynamiques, qui sont décrites dans [60]. Elles ont l'avantage d'être efficaces et permettent d'optimiser la quantité de mémoire utilisée pour stocker des bits. En effet, la taille occupée en mémoire est le ratio entre le nombre total de bits à stocker dans la table et le nombre de bits contenus dans 1 octet, c'est-à-dire 8 bits. Par exemple si la table dynamique contient B bits, alors elle occupe $\lceil \frac{B}{8} \rceil$ (partie entière supérieure) octets en mémoire.
Afin de mesurer l'impact de l'utilisation de cette structure de données, plusieurs segmentations ont été effectuées sur des images de taille croissante en utilisant 3 stratégies pour stocker la localisation des segments. La première consiste à stocker les coordon-

nées de tous les pixels internes de chaque segment en utilisant le type *long unsigned int*. Cette stratégie nécessite 64 bits en mémoire pour stocker les coordonnées d'un pixel. La seconde stratégie consiste à stocker les coordonnées des pixels situés sur les contours des segments toujours en utilisant le type *long unsigned int*. Elle nécessite également 64 bits en mémoire pour stocker les coordonnées de chaque pixel. Enfin, la troisième stratégie consiste à stocker la liste des déplacements codés sur 2 bits le long des contours des segments.
Trois images de taille croissante ont été segmentées avec l'algorithme générique utilisant le critère d'homogénéité Baatz & Schäpe. Pour chacune des images, les 3 stratégies pour stocker la localisation des pixels ont été utilisées. Les images originales ainsi que les images segmentées sont représentées dans la figure 3.10.

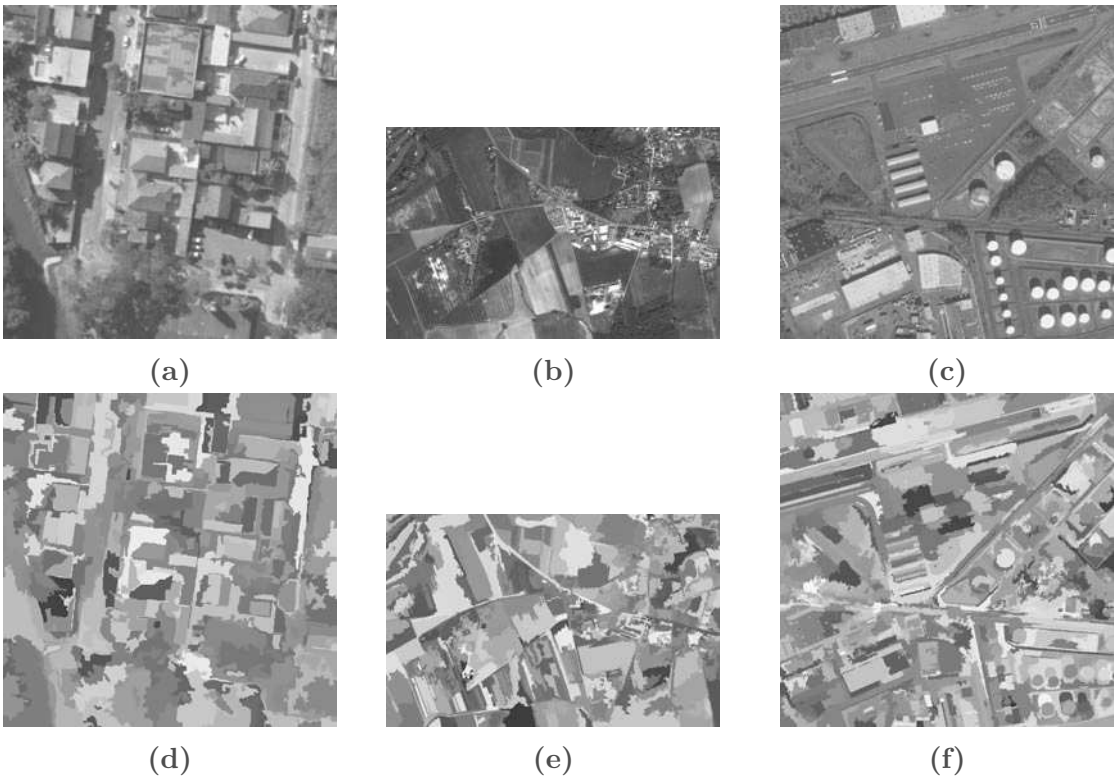
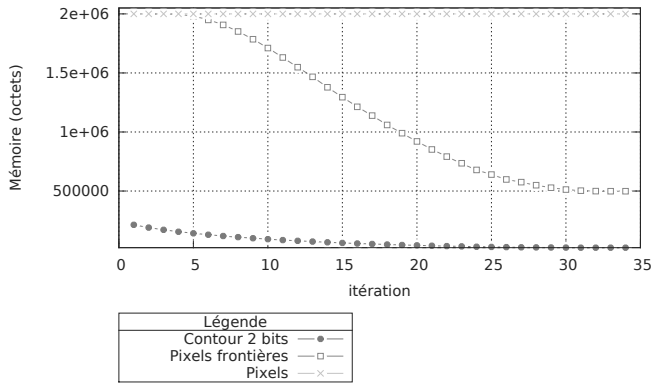


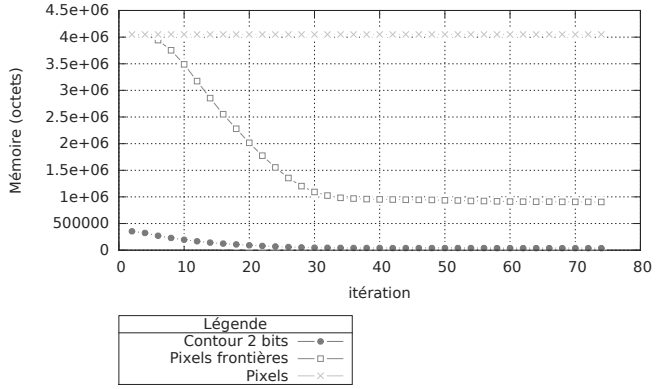
Figure 3.10 – Segmentation avec l'algorithme générique de trois images de taille croissante pour analyser les stratégies utilisées pour représenter les coordonnées des pixels : (a) 500×500 pixels, (b) 893×567 pixels, (c) 1000×1000 pixels.

La figure 3.11 représente pour chaque image l'évolution de la mémoire en octets utilisée pour chaque stratégie en fonction du nombre d'itérations.

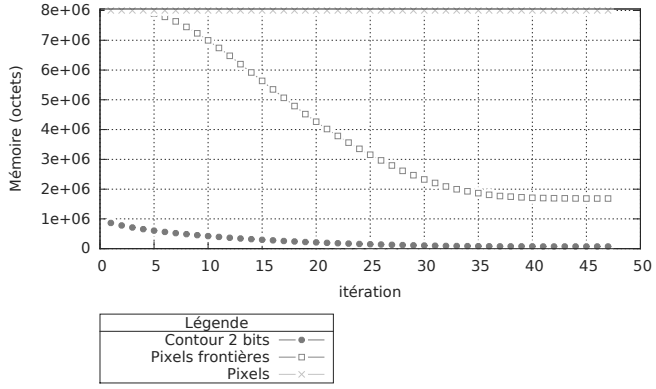
Nous remarquons à travers ces résultats que la composition de l'image influe sur la vitesse de réduction de l'usage de la mémoire pour les stratégies consistant soit à utiliser les pixels frontières (stratégie 2), soit le contour sur 2 bits (stratégie 3). En effet,



(a)



(b)



(c)

Figure 3.11 – Évolution de la mémoire en octets utilisée pour chaque stratégie de représentation de la localisation des pixels en fonction du nombre d'itérations

plus les objets d'intérêt composant l'image sont petits et plus ce facteur de réduction

est faible. Par exemple, l'image 3.10b contient de plus grandes zones homogènes et par conséquent la vitesse de réduction des stratégies 2 et 3 est plus importante (figure 3.11b). Cependant, nous notons que les facteurs de réduction entre la stratégie 3 et la stratégie 2 sont corrélés pour chaque image. Ce résultat est logique, car le nombre de pixels frontières et le nombre de déplacements le long du contour sont également corrélés. Nous pouvons conclure qu'utiliser la représentation des contours permet de réduire significativement la quantité de mémoire nécessaire pour stocker les coordonnées des segments. Nous utilisons logiquement cette stratégie par la suite.

3.3 Description de l'algorithme GRM et analyse de la complexité

Rappelons que l'idée de l'algorithme GRM est de proposer un squelette d'exécution commun à toutes les méthodes de segmentation par fusion de régions. Ce squelette prend en charge les opérations sur le graphe stockant les segments ainsi que la gestion du contour et du voisinage de chaque segment tout au long de la procédure de segmentation. Ces différentes opérations sont décrites par la suite.

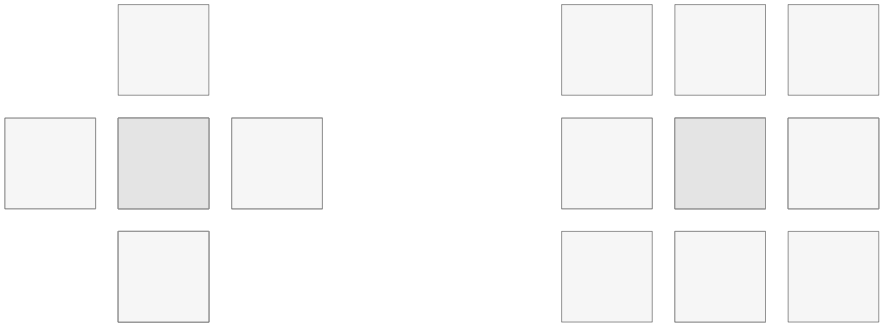
3.3.1 Initialisation des segments dans l'image

La première étape consiste à construire les segments du graphe à partir de l'image. Dans l'algorithme GRM, chaque pixel représente un segment initial. L'attribut id de chaque segment est initialisé aux coordonnées du pixel dans l'image. Concrètement, si (x, y) sont les cordonnées du pixel dans une image de largeur w alors $id = y * w + x$. De cette manière, chaque segment possède un identifiant unique. La boîte englobante de chaque segment correspond aux dimensions du pixel (égales à 1). Le contour du segment est composé de 4 déplacements le long du pixel comme illustré par la figure 3.5.

La seconde étape consiste à construire les liens d'adjacence des segments dans l'image. L'algorithme GRM met à disposition 2 types de connectivité : "4-connexité" et "8-connexité". Ces 2 connectivités sont illustrées par la figure 3.12.

Le voisinage est calculé pour chaque segment dans le graphe. Si dans une direction donnée un voisin existe, alors une arête est créée entre les 2 segments. Une fois cette étape terminée, l'algorithme GRM rend la main à l'utilisateur pour faire appel à une fonction d'initialisation des attributs spécifiques.

En conclusion, la complexité en espace de la phase d'initialisation est $O(N+M+E)$ où N est le nombre de segments dans le graphe, M le nombre total de déplacements autour des contours des segments et E le nombre total d'arêtes. Le raisonnement suivant ne dépend pas du choix de la connexité des pixels. Dans la suite, nous utilisons un voisinage 4-connexe pour les exemples et les figures. Ainsi, chaque segment contient au plus 4 segments adjacents. Puisqu'un segment initial contient un seul pixel, alors il contient 4 déplacements. La complexité en temps pour construire le graphe est donc linéaire $O(N)$.



(a) “4-connectivité” **(b)** “8-connectivité”

Figure 3.12 – 2 types de connectivité disponibles dans l’algorithme GRM

3.3.2 Calcul des coûts de fusion

À chaque itération, la première étape consiste à mettre à jour les coûts de fusion entre les paires de segments adjacents dans le graphe. Elle correspond à l’étape 1 dans la figure 2.1b. Le coût de fusion doit être mis à jour pour une paire de segments adjacents si l’un des 2 segments a fusionné à l’itération précédente et si le coût de fusion n’est pas à jour.
Les arêtes de chaque segment du graphe sont parcourues et le coût de fusion est calculé, si nécessaire. Les complexités en temps et en espace pour calculer le coût de fusion sont inconnues puisque cette fonction est définie par l’utilisateur. Elles sont notées respectivement O_{ft} et O_{fm}.
Pour une paire de segments adjacents S_i et S_j, le calcul du coût de fusion $c_{i,j}$ n’est effectué qu’une seule fois puisque $c_{i,j} = c_{j,i}$. En effet, nous considérons que la mesure de similarité est symétrique. Une fois $c_{i,j}$ calculé et affecté à l’arête pointant vers S_j, il est nécessaire de trouver l’arête pointant vers S_i pour lui affecter $c_{j,i}$. Trouver l’arête entre le segment adjacent et le segment courant a une complexité en temps $O(\bar{E})$ où $\bar{E}$ représente le nombre d’arêtes moyen par segment dans le graphe.
Lors du parcours des arêtes d’un segment, nous retenons l’arête pour laquelle le coût de fusion est minimum. Après avoir calculé le coût de fusion pour toutes les arêtes du segment courant, nous échangeons la première arête du segment avec celle dont le coût est minimum. Ainsi, lors de la phase de fusion des segments, la première arête de chaque segment correspondra à celle pointant vers le segment adjacent le plus similaire selon le critère d’homogénéité.
Nous pouvons conclure que la complexité en temps de cette opération est $O\left(N \times \bar{E} \times \left(O_{ft} + \bar{E}\right)\right)$ et la complexité en espace est $O(N + E + M + O_{fm})$.

3.3.3 Mise à jour du contour

Soit la paire de segments S_i et S_j qui doivent être fusionnés. La mise à jour du contour est décrite dans la section 3.2.4. Elle consiste à générer la liste des coordonnées des pixels situés sur les frontière des deux contours des segments et à parcourir cette liste pour recréer le nouveau contour. Nous notons $\bar{M}$ le nombre moyen de déplacements par segment dans le graphe. La génération de la liste des coordonnées des pixels consiste à parcourir la liste des déplacements de chaque segment. La complexité en temps de cette opération est $O(2\bar{M})$. Nous notons $\bar{P}$ le nombre maximum de pixels situés sur la frontière d'un segment dans le graphe. La complexité en espace est $O(2\bar{P})$. La création du nouveau contour consiste à parcourir l'ensemble des pixels frontière de la liste. La complexité en temps est alors $O(2\bar{P})$.
Nous pouvons donc conclure que la complexité en temps de cette opération est $O(2\bar{M} + 2\bar{P})$ et la complexité en espace est $O(N + E + M + 2\bar{P})$ car il faut stocker en plus la liste des pixels générée.

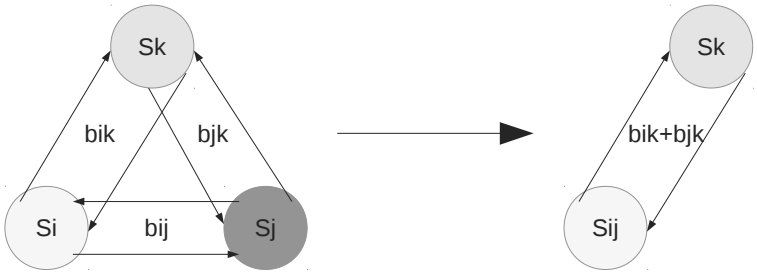
3.3.4 Mise à jour du voisinage

Soit une paire de segments adjacents S_i et S_j qui doivent être fusionnés. Nous supposons que S_j fusionne dans S_i pour former $S_{i,j}$. Cette opération consiste à modifier le voisinage des segments adjacents de S_j et celui de S_i tout en mettant à jour la valeur $frontiere$ notée $b_{i,j}$ entre S_i et S_j. La figure 3.13 illustre l'ensemble des configurations possibles.

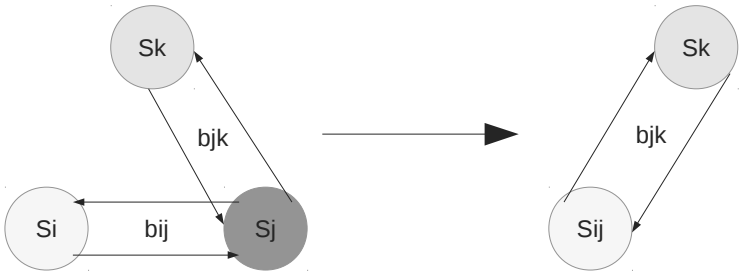
Les segments adjacents de S_j sont parcourus. Soit S_k un segment adjacent de S_j et la valeur $b_{j,k}$ entre S_j et S_k. La première étape consiste à déterminer si S_k est aussi un segment adjacent de S_i.
Si c'est le cas (figure 3.13a), alors le lien d'adjacence est supprimé entre S_k et S_j et $b_{i,k}$ est incrémentée par la valeur $b_{j,k}$.
Sinon (figure 3.13b), le lien d'adjacence est supprimé entre S_k et S_j et un lien d'adjacence est ajouté entre S_i et S_k avec la valeur $b_{i,k} = b_{j,k}$.
La complexité en espace de cette opération est $O(N + M + E)$ et la complexité en temps est $O(\bar{E}^2)$ car les segments adjacents de S_j sont parcourus et pour chaque segment adjacent S_k, la recherche d'une arête pointant vers S_i est effectuée.

3.3.5 Suppression des segments fusionnés

Une fois l'itération effectuée, les segments ayant fusionné sont supprimés du graphe. La structure de données utilisée pour stocker les segments est un tableau dynamique. Nous avons introduit dans le Chapitre 1 Section 1.4, un algorithme permettant de supprimer des éléments dans un tableau dynamique avec une complexité en temps linéaire $O(N)$. La procédure pour supprimer les segments du graphe est la même que celle illustrée dans la figure 1.6. Les segments qui ont fusionné dans d'autres segments sont déplacés à la fin du tableau dynamique et sont supprimés avec une complexité en temps



(a) S_i et S_j possède un segment adjacent commun S_k.



(b) S_j possède un segment adjacent S_k qui n'a pas de lien d'adjacence avec S_i.

Figure 3.13 – Ensemble des configurations possibles du voisinage lors de la fusion de S_j dans S_i.

constante $O(1)$.
La complexité en temps de cette opération est donc $O(N)$ et celle en espace $O(N+M+E)$.

3.3.6 Algorithme GRM

Après avoir analysé chaque étape de l'algorithme GRM, déterminons les complexités en espace et en temps de celui-ci en nous basant sur le pseudo-algorithme 3.1

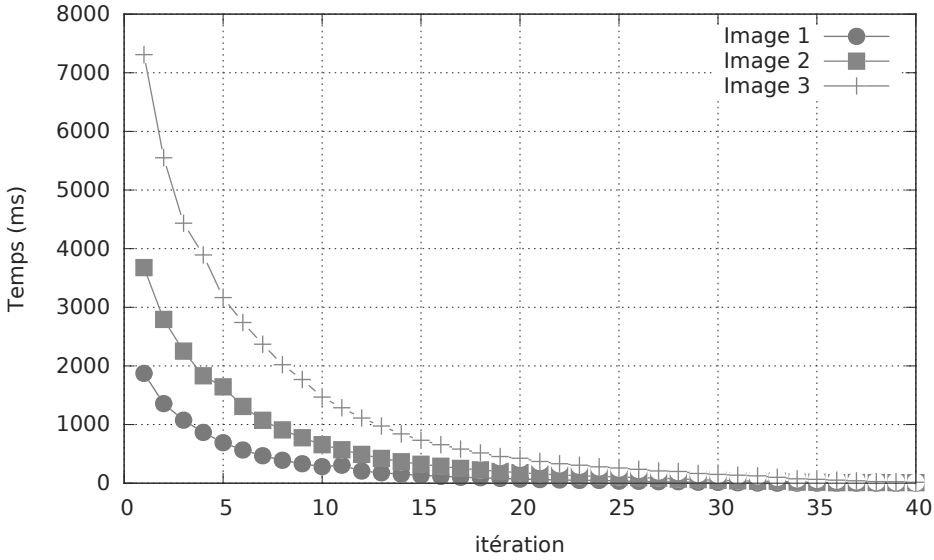
Récapitulons tout d'abord les complexités de chaque étape de l'algorithme GRM. La phase d'initialisation du graphe (ligne 2) a une complexité en espace $O(N+M+E)$ et une complexité en temps $O(N)$. Le calcul des coûts de fusion (ligne 4) a une complexité en temps $O\left(N \times \bar{E} \times \left(O_{ft} + \bar{E}\right)\right)$ et une complexité en espace $O(N+M+E+O_{fm})$. Dans la ligne 5, nous entrons dans une boucle qui parcourt tous les segments, cela nécessite ainsi N itérations. La mise à jour des attributs spécifiques est inconnue et dépend de l'utilisateur. Nous notons sa complexité en temps O_{ut} et sa complexité en espace O_{um}. La mise à jour du contour (ligne 8) a une complexité en temps $O(2\bar{M}+2\bar{P})$ et une complexité en espace $O(N+M+E+2\bar{P})$. La mise à jour du voisinage (ligne 9) a une complexité en temps $O(\bar{E}^2)$ et une complexité en espace $O(N+M+E)$. Par conséquent, la complexité en temps de la boucle est $O\left(N \times \left(O_{ut} + O(2\bar{M}+2\bar{P}) + O(\bar{E}^2)\right)\right)$ et la complexité en

espace est $O(M+N+E+\max\{O_{um}, O(2\bar{P})\})$. Enfin, la supression des segments fusionnés a une complexité en temps $O(N)$ et une complexité en espace $O(N+M+E)$.

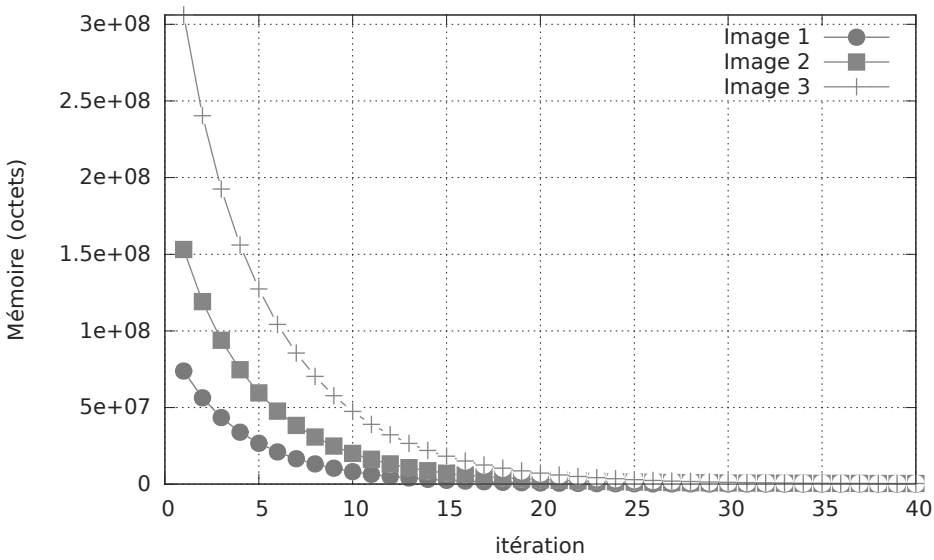
Le nombre de segments, d'arêtes et de déplacements le long des contours diminue au fil des itérations, ce qui induit une accélération du temps d'exécution de l'algorithme GRM et une réduction de l'espace mémoire nécessaire. Cependant, une attention particulière doit être portée sur le développement des fonctions de coût et de mise à jour des attributs spécifiques car elles peuvent avoir un impact majeur sur la complexité en temps de la procédure de segmentation. En effet, le nombre de segments dans l'image est facteur de ces complexités.

Pour illustrer l'accélération du temps d'exécution au fil des itérations et la réduction de la capacité de la mémoire nécessaire, nous avons considéré les 3 images représentées dans la figure 3.10. Chacune de ces images a été segmentée avec l'algorithme GRM et le critère de Baatz & Schäpe en mesurant à chaque itération le temps d'exécution ainsi que la quantité de mémoire nécessaire. Les graphes montrant l'évolution du temps d'exécution (en millisecondes) et de la quantité de mémoire utilisée (en octets) en fonction du nombre d'itérations sont représentés dans la figure 3.14.

L'algorithme GRM est conçu en supposant que l'image ainsi que le graphe des segments puissent être stockés en mémoire interne tout au long de la procédure de la segmentation. Avec l'arrivée des missions d'observation de la Terre à très haute résolution, cette hypothèse n'est plus vraie. Par exemple, la constellation des satellites Pléiades fournissent des images THR de taille 40000×40000 pixels avec 4 bandes spectrales. Stocker l'image complète et le graphe des segments initiaux devient impossible même sur les ordinateurs actuels qui possèdent quelques dizaines de gigaoctets en mémoire interne. La seule solution est alors de diviser l'image en tuiles de telle façon que chaque tuile et son graphe de segments correspondant puissent être stockés en mémoire interne. Cependant, nous allons voir que cette stratégie a un impact négatif sur la qualité des segments résultants. Par conséquent, une solution doit être apportée pour garantir un résultat identique même lorsqu'une stratégie de tuilage est effectuée.
Dans les chapitres suivants, nous analysons l'impact du tuilage lors de l'utilisation de l'algorithme GRM. Après avoir identifié les points critiques liés au tuilage, nous étudions le développement d'une solution stable.



(a) Évolution du temps d'exécution en fonction du nombre d'itérations.



(b) Quantité de mémoire utilisée en fonction du nombre d'itérations

Figure 3.14 – Évolution du temps d'exécution et de la quantité de mémoire utilisée en fonction du nombre d'itérations

Chapitre 4

Stabilisation de l'algorithme GRM

4.1 Introduction

Les récentes missions satellite d'observation de la Terre telles que Quickbird, WorldView, GeoEye et Pléiades fournissent des images à très haute résolution. Ces images permettent d'obtenir une information très détaillée de la surface terrestre utile pour des applications telles que la surveillance environnementale, la gestion des ressources ou la surveillance urbaine. La constellation des satellites Pléiades permet d'acquérir des images avec un échantillonage au sol de 50 cm par pixel et une couverture spatiale de 400 km^2. Par conséquent, une image Pléiades a une taille de 40000 $\times$ 40000 pixels, ce qui représente un large volume de données à exploiter. Nous rappelons que les caractéristiques des satellites Pléiades sont décrites dans l'annexe A. Segmenter de telles images réprésente un réel enjeu à cause de la limitation de la quantité de mémoire interne disponible dans les ordinateurs.

Pour surmonter cet obstacle, la solution adoptée jusqu'à présent [6] est de diviser l'image en tuiles [1] et traiter chaque tuile séparément. Nous nommons cette opération le tuilage. Un algorithme est facilement parallélisable ("embarrassingly parallel") si l'ensemble des données en entrée peut être décomposé en sous-ensembles de données pouvant être traités séparément sans nécessiter de communication. Un exemple d'algorithme facilement parallélisable est un filtre gaussien utilisant une fenêtre de convolution à taille fixe. Un tel algorithme est utilisé pour éliminer le bruit dans une image. Un exemple de

1. Une tuile est une portion rectangulaire extraite de l'image dont les côtés sont parallèles aux axes de l'image.

fenêtre de convolution de taille 5×5 pixels avec $\sigma = 0.625$ est représenté ci-dessous :

$$\frac{1}{250}\begin{bmatrix} 0.03 & 0.16 & 5.98 & 0.16 & 0.03 \\ 0.16 & 7.7 & 27.80 & 7.7 & 0.16 \\ 5.98 & 27.8 & 100 & 27.8 & 5.98 \\ 18 & 80 & 132 & 80 & 18 \\ 0.03 & 0.16 & 5.98 & 0.16 & 0.03 \end{bmatrix}$$

Lorsque l'image est divisée en tuiles, le traitement des pixels sur la bordure des tuiles est problématique puisque des pixels situés dans la fenêtre de voisinage sont manquants. La stratégie est alors de considérer autour de la tuile une marge supplémentaire de 2 pixels . Elle est illustrée dans la figure 4.1.

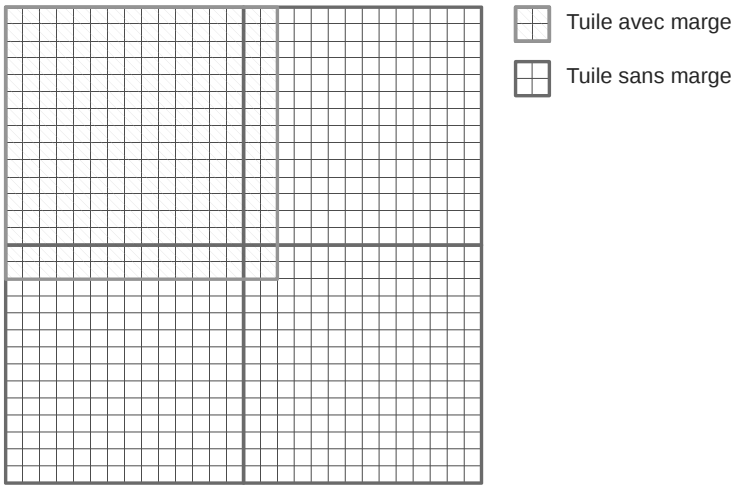


Figure 4.1 – Tuile avec sa marge pour assurer un résultat identique lors de l'application d'un filtre gaussien.

Cette solution peut s'appliquer à n'importe quel traitement par pixel ou n'importe quel traitement par convolution utilisant une fenêtre de voisinage statique et permet de garantir un résultat identique à celui obtenu sans tuilage.

Cependant, pour des algorithmes non réguliers comme la plupart des méthodes de segmentation, la marge autour de chaque tuile ne peut pas être déterminée à l'avance. En effet, des interactions entre des pixels spatialement éloignés dans l'image peuvent exister. Nous allons voir que les méthodes de segmentation et en particulier l'algorithme GRM ne sont pas compatibles avec le tuilage. Dans cette thèse, l'objectif des travaux sur la segmentation est de proposer une solution utilisant le tuilage qui garantit des segments résultants identiques à ceux que nous aurions obtenus si l'image pouvait être segmentée sans tuilage.

Dans la prochaine section, nous allons étudier l'impact du tuilage lors de l'application d'une segmentation avec l'algorithme GRM. Enfin, la dernière section de ce chapitre

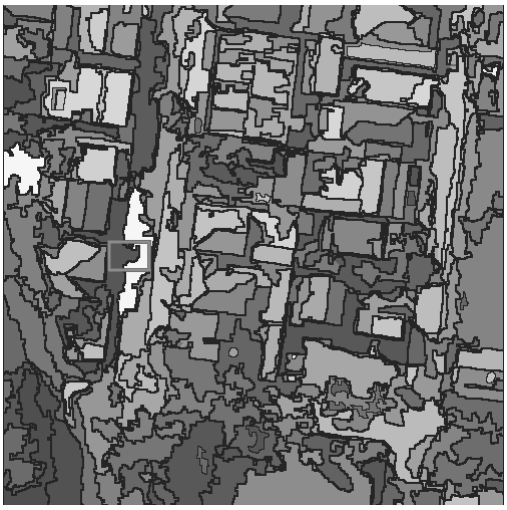
récapitule les différentes solutions proposées combinant le tuilage avec un algorithme de segmentation.

4.2 Impact du tuilage

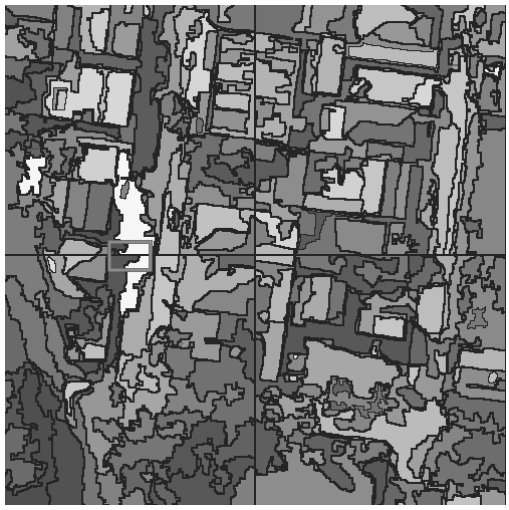
Afin d'illustrer visuellement l'impact du tuilage sur les segments résultants avec l'algorithme GRM, nous proposons une première expérience. Le critère utilisé dans cette section est celui de Baatz & Schäpe décrit dans la section 2.3. Pour ce critère, nous configurons le paramètre d'échelle $s = 60$, le poids pour l'importance relative de l'hétérogénéité spectrale $w_c = 0,5$ et celui pour l'importance relative du degré de compacité $w_s = 0,5$. Une image THR de taille 500×500 pixels fournie par le satellite Ikonos est considérée et représentée dans la figure 4.2.

Figure 4.2 – Image utilisée pour les expériences

L'image est d'abord segmentée en une seule fois et le résultat de la segmentation est illustré par la figure 4.3a. Pour la deuxième segmentation, l'image est préalablement divisée en 4 tuiles de taille 250×250 pixels. Chaque tuile est ensuite segmentée et l'image résultante est reconstituée en regroupant les tuiles segmentées. Le résultat de cette segmentation est illustré par la figure 4.3b. Une première analyse visuelle met en évidence la présence d'artefacts sur les bordures de la tuile (figure 4.3c) mais aussi la présence de segments différents à l'intérieur de la tuile (figure 4.3d). Afin de confirmer nos observations, nous proposons une seconde expérience qui utilise une métrique de comparaison pour identifier et mesurer les différences entre les 2 segmentations.



(a) GT



(b) TS

(c) artefacts

(d) segments différents

Figure 4.3 – Analyse visuelle de l'impact du tuilage sur les segments résultants. Dans la figure 4.3b, les frontières des tuiles sont délimitées par des lignes continues noires. Le rectangle rouge met en évidence la présence d'un artefact sur une bordure commune entre 2 tuiles adjacentes. La figure 4.3c est un zoom sur l'artefact contenu dans le rectangle rouge. La figure 4.3d est un zoom mettant en évidence un segment différent à l'intérieur d'une tuile qui n'existe pas dans la segmentation de référence.

Il existe de nombreuses métriques pour comparer deux résultats de segmentation et une revue de celles-ci est faite dans [61]. Nous avons choisi d'utiliser la méthode de Hoover proposée dans [62] pour comparer les segmentations. Cette méthode compare deux segmentations dont l'une est la segmentation de référence, notée GT, et l'autre est la segmentation test, notée TS. Elle consiste à associer un ou plusieurs segments dans GT avec un ou plusieurs segments dans TS. Un segment dans GT est noté S_i^1 et un segment dans TS, S_j^2. Il existent 4 types d'association qui sont appelés des instances de Hoover. Ces instances sont basées sur la matrice de confusion O qui est définie de la façon suivante :

$$O_{ij} = |S_i^1 \cap S_j^2| \text{ pour } (S_i^1, S_j^2) \in \text{GT} \times \text{TS} \tag{4.1}$$

Soit t un seuil de recouvrement défini par l'utilisateur tel que $0.5 \leq t \leq 1$. L'instance

RC représente une association entre 2 segments S_i^1 et S_j^2 si :

$$\begin{cases} O_{ij} \geq t \times |S_i^1| \\ O_{ij} \geq t \times |S_j^2| \end{cases} \tag{4.2}$$

L'instance RF est détectée si S_i^1 est fragmentée en segments S_*^2 dans TS avec $\{S_*^2 = S_{j_k}^2 \in TS, k \in [1, m]\}$ et :

$$\begin{cases} \forall k \in [1, m], & O_{ij_k} \geq t \times |S_{j_k}^2| \\ & \sum_{k=1}^{m} O_{ij_k} \geq t \times |S_i^1| \end{cases} \tag{4.3}$$

L'instance RA est détectée si S_j^2 est fragmentée en segments S_*^1 dans GT avec $\{S_*^1 = S_{i_k}^1 \in GT, k \in [1, m']\}$ et :

$$\begin{cases} \forall k \in [1, m'], & O_{i_k j} \geq t \times |S_{i_k}^1| \\ & \sum_{k=1}^{m'} O_{i_k j} \geq t \times |S_j^2| \end{cases} \tag{4.4}$$

Enfin l'instance RM est détectée si les segments S_i^1 et S_j^2 n'appartiennent à aucune des 3 instances décrites précédemment. La méthode de Hoover retourne la proportion des segments appartenant à chaque instance. Les valeurs des proportions sont comprises entre 0 et 1. Dans l'expérience suivante, $t = 1$, ce qui signifie que nous sommes intrangiseants sur l'égalité des segments. Ainsi S_i^1 et S_j^2 appartiennent à RC si $S_i^1 \cap S_j^2 = S_j^2 = S_i^1$. Lorsque les deux segmentations sont identiques, nous devons avoir $RC = 1$, $RF = 0$, $RA = 0$ et $RM = 0$.

Le protocole de l'expérience est celui utilisé dans [63] pour mesurer la stabilité de plusieurs familles d'algorithmes de segmentation. Dans un premier temps, l'image est segmentée en une seule fois puis considérée comme la segmentation de référence. Dans un second temps, une tuile est extraite de l'image puis segmentée. Elle est notée TS. La zone correspondant à la tuile est aussi extraite de la segmentation de référence et est notée GT. Cette procédure est effectuée pour différentes tailles de tuile en augmentant la largeur de la couronne w. L'ensemble de la procédure est illustré par la figure 4.4.

Cette procédure a été effectuée pour les heuristiques de décision BF et LMBF. L'évolution des instances de Hoover en fonction de la largeur de la couronne w est représentée par les graphes dans la figure 4.5.

D'après les graphes, nous pouvons conclure que l'heuristique LMBF est moins sensible au tuilage que l'heuristique BF. En effet, avec LMBF la proportion de segments appartenant à RC est toujours au-dessus de celle avec BF, et cela pour chaque valeur de w. Cette observation appuie notre choix quant à l'utilisation de LMBF pour l'algorithme GRM. Par contre, pour chaque heuristique, les segmentations GT et TS ne sont jamais identiques ($RC < 1$) excepté pour $w = 0$. Nous pouvons ainsi conclure que le tuilage modifie le résultat final. Nous notons également que cette modification s'accentue pour des valeurs croissantes de w, sachant qu'augmenter w revient à considérer des tuiles de plus en plus petite dans l'image.

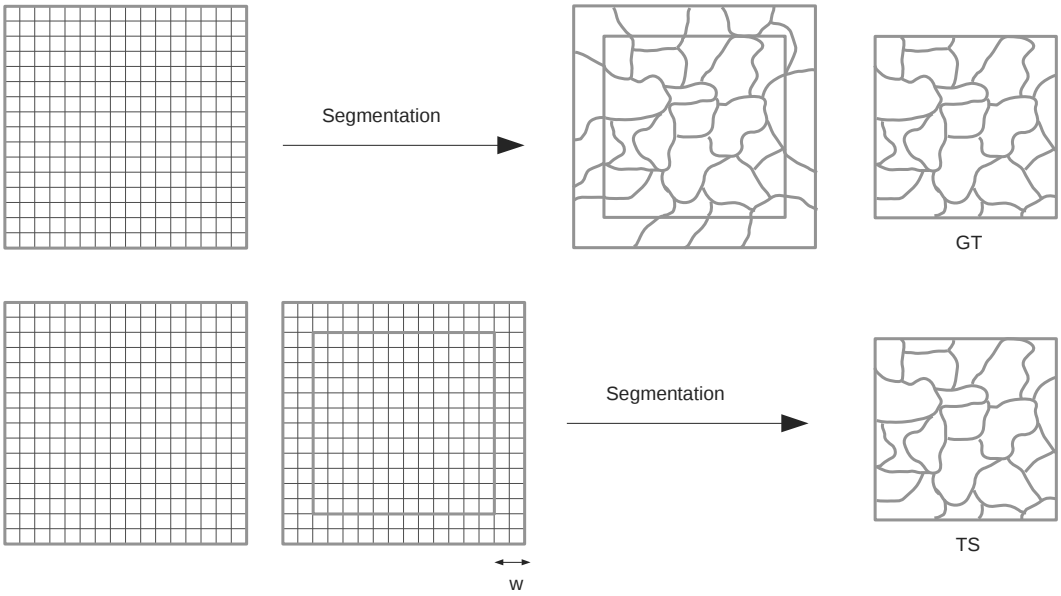


Figure 4.4 – Protocole pour la comparaison des segmentations GT et TS. Cette procédure est répétée pour différentes valeurs de w.

Dans la prochaine section, nous décrivons les solutions proposées pour résoudre la présence d'artefacts sur les bordures communes des tuiles (figure 4.3c) et supprimer l'impact du tuilage sur les segments résultants.

4.3 Passage à l'échelle des méthodes de segmentation : état de l'art

Plusieurs approches ont été étudiées pour minimiser l'impact du tuilage sur la qualité des segments résultants.

Dans [64, 65], l'auteur introduit la notion de segment contagieux pour éviter la présence d'artefacts sur les bordures communes des tuiles adjacentes. À l'étape initiale, les segments situés sur les bordures communes des tuiles sont marqués comme contagieux. Au fil des itérations, si un segment non contagieux fusionne avec un segment contagieux alors il devient lui-même contagieux. Enfin, lorsque deux segments sont contagieux, ils ne peuvent pas fusionner. Ainsi, avec cette stratégie bloquante, la solution empêche l'apparition d'artefacts sur les bordures des tuiles segmentées. Cependant, cette approche n'est pas appropriée dans la situation où il y a beaucoup de segments contagieux car la procédure de fusion s'arrête prématurément entraînant une sur-segmentation de l'image.

Dans [52], les auteurs proposent une solution générique à tous les algorithmes de fusion de régions pour éviter des incompatibilités entre les segments situés de part et

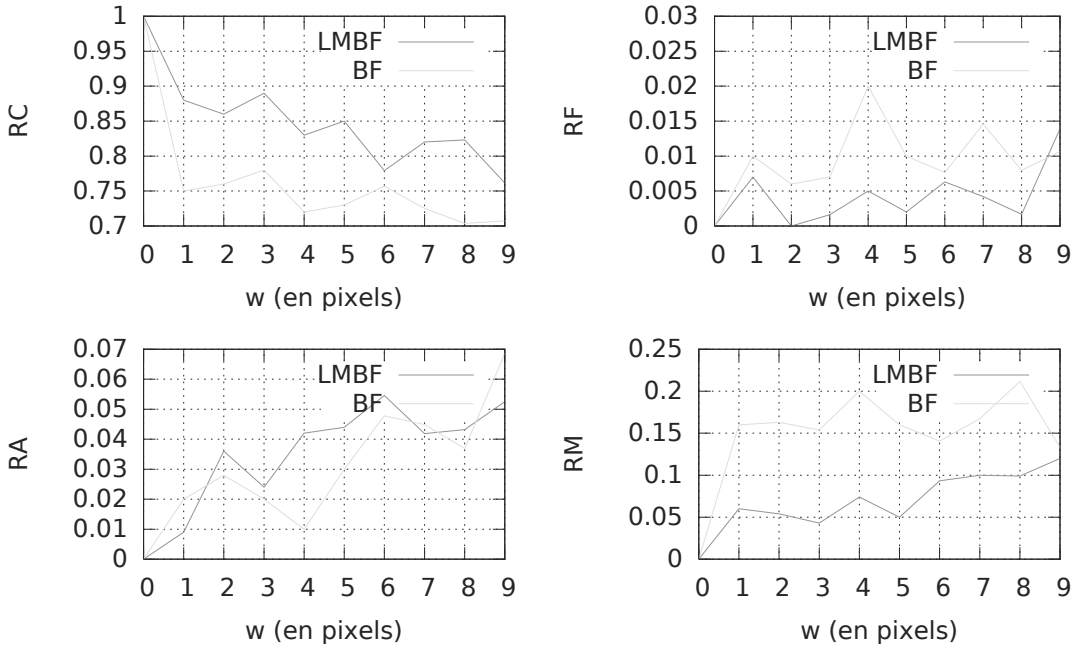


Figure 4.5 – Évolution des instances de Hoover en fonction de la largeur de la couronne w. Les segmentations ont été comparées en utilisant 2 heuristiques de décision pour la fusion des segments : BF et LMBF.

d'autre des bordures communes des tuiles adjacentes. Le critère d'homogénéité utilisé dans leurs travaux est le critère Full Lambda Schedule décrit dans la section 2.3. Les auteurs émettent l'hypothèse que la présence d'artefacts sur les bordures des tuiles est due à un phénomène local. C'est pourquoi, ils proposent de diviser l'image en tuiles avec des zones de recouvrement sur les bordures. L'inconvénient de cette solution est qu'il peut y avoir au moins deux segmentations différentes d'une même zone. Pour pallier ce problème, les auteurs proposent plusieurs règles en supposant qu'un segment situé près du centre de la tuile a plus de chance d'être correct qu'un segment situé près de la bordure. La première hypothèse est de considérer que les segments inclus totalement ou partiellement dans les zones de non recouvrement des tuiles sont corrects. L'idée est ainsi de segmenter à nouveau les segments qui sont totalement inclus dans la zone de recouvrement dans une deuxième étape en considérant les contours des segments corrects de la première étape comme des frontières naturelles de l'image. Déterminer la taille correcte de la zone de recouvrement est problématique car elle dépend de la nature des objets d'intérêt composant l'image. Ainsi, avec cette méthode, il n'y a aucune garantie de l'absence d'artefacts dans les tuiles.

Dans [66], les auteurs proposent une méthode pour détecter les réseaux routiers dans les images satellite en utilisant une méthode de segmentation composée de plusieurs étapes. La première étape consite à utiliser une méthode de segmentation globale basée

sur un graphe [67] pour former des segments dans l'image. Une étape de fusion de régions est ensuite effectuée pour détecter les routes à partir de considérations géométriques. Dans leurs travaux, les images satellites ont d'abord été divisées en tuiles de 200×200 pixels qui sont ensuite segmentées séparément. La procédure de fusion de régions est finalement appliquée pour tous les segments dans chaque tuile sans se soucier des segments situés sur les bordures des tuiles entraînant ainsi la présence d'artefacts.

Dans [68], les auteurs proposent une solution pour l'algorithme de segmentation [67] basée sur la recherche d'arbres minima dans un graphe. Au lieu de diviser l'image en tuiles, l'algorithme est conçu pour permettre la parallélisation sans couper les objets localisés sur les bordures des tuiles. Pour ce faire, des algorithmes de recherche d'arbres minima sont exécutés en parallèle sur chacune des tuiles et détectent si une arête se situe à une bordure de la tuile. Dans cette situation, l'exécution de l'arête est repoussée jusqu'à que les recherches d'arbres minima dans les tuiles adjacentes soient aussi terminées. Ces arêtes retardées sont ensuite exécutées. En pratique, les auteurs prétendent que seulement 5% des arêtes sont retardées et que la solution permet d'obtenir un résultat de segmentation sans artefacts. Cependant, cette solution est spécifique à l'algorithme proposé dans [67] qui est un algorithme basé seulement sur l'intensité spectrale des pixels et ne peut être appliquée aux méthodes de segmentation par fusion de régions.

Dans [69], les auteurs proposent une solution similaire à celle proposée pour les segments contagieux. L'idée est de diviser l'image en tuiles et de segmenter chaque tuile en parallèle avec un algorithme de fusion de régions utilisant le critère de Baatz & Schäpe. Chaque itération est exécutée en parallèle sur chaque tuile et l'exécution des segments situés sur les bordures communes des tuiles est retardée à la fin de l'itération. Une fois l'itération accomplie sur l'ensemble des tuiles, une procédure de segmentation séquentielle est exécutée pour traiter les segments sur les bordures des tuiles. Le premier inconvénient de cette méthode est que le nombre de segments à traiter séquentiellement peut être potentiellement important. Par conséquent, le gain obtenu par la parallélisation peut être remis en question. Enfin, cette méthode ne garantit pas des segments résultants identiques comparés à ceux obtenus en segmentant l'image sans tuilage.

Dans [70], l'auteur propose l'algorithme RHSEG qui est une solution approximative de l'algorithme hiérarchique par fusion de régions HSEG [71]. L'auteur indique que la stratégie bloquante dans [64, 65] induit un arrêt prématuré de la procédure de segmentation. Pour résoudre ce problème, il propose uns stratégie "Split & Merge" à la fin de chaque itération pour reconsidérer les segments contagieux dans leurs états initiaux et appliquer une procédure de fusion de régions itérative sur ces segments. L'objectif est de réaffecter des pixels contenus dans un segment contagieux à une autre région avec laquelle ils sont plus similaires. Un inconvénient de cette méthode est qu'elle crée des segments qui ne sont pas connectés spatialement. Ceci est problématique si une opération d'étiquetage doit être effectuée, car elle induit un surcoût éventuel au niveau du temps de calcul. De plus, cette méthode a été conçue dans l'optique d'une segmentation hiérarchique de l'image mais pas sur la problématique de traitement de grandes images ne pouvant être stockées dans la mémoire interne.

Dans [72], les auteurs proposent de diviser l'image en tuiles dont les bordures suivent

le gradient de l'image. De cette manière, ils espèrent que les bordures s'adaptent aux contours des segments situés dans la zone de découpage. Afin de trouver le gradient maximum dans la zone de découpage, ils utilisent un tampon de pixels d'une certaine largeur. L'inconvénient est que cette solution n'est pas parfaite pour les segments dont la taille est supérieure à la largeur de ce tampon. Ces segments seront alors découpés entraînant ainsi des artefacts sur les bordures des tuiles. Un autre inconvénient est que cette méthode n'est pas généralisable à n'importe quel critère de fusion dans la mesure où certains critères ne dépendent pas du gradient local.

Dans [73], l'auteur propose d'identifier les bordures des tuiles avec les contours naturels des segments. Le critère est utilisé lors de la procédure de segmentation et est basé sur une analyse colorimétrique des segments adjacents et du segment courant. Cependant, aucune garantie n'est donnée sur le fait que les contours des segments qui représentent en réalité les bordures des tuiles soient tous détectés. Là encore, nous retrouvons l'inconvénient que la solution n'est pas généralisable à tous les critères de fusion car certains critères n'utilisent pas le gradient local.

Dans [6], l'image est divisée en tuiles et chaque tuile est ensuite segmentée indépendamment. Un traitement supplémentaire consiste à utiliser un critère topologique pour fusionner les segments de part et d'autre des bordures communes entre les tuiles adjacentes. L'idée est de fusionner deux segments si la surface de contact de leur contour est supérieure à un seuil défini par l'utilisateur. Bien que cette solution soit générique à n'importe quelle méthode de segmentation, elle ne garantit pas la suppression de tous les artefacts, puisqu'un segment peut avoir plusieurs surfaces de contact avec des segments, qui soient inférieures au seuil.

Dans [74], l'auteur propose un cadre pour segmenter de larges images en utilisant le tuilage. Dans un premier temps, l'image est divisée en tuiles sans zone de recouvrement. Chaque tuile est ensuite segmentée avec le critère Full Lambda Schedule décrit dans la section 2.3. Une fois les tuiles segmentées, l'image résultante est représentée sous la forme d'une grille de tuiles. L'étape suivante consiste à fusionner les tuiles suivant les lignes de la grille et ensuite suivant les colonnes de la grille. La fusion des tuiles suivant les lignes permet d'obtenir une grille d'une seule ligne et de plusieurs colonnes. Les tuiles sont ensuite fusionnées suivant les colonnes pour obtenir l'image segmentée finale. Notons que le processus peut être effectué dans l'ordre inverse. Il serait d'ailleurs intéressant d'analyser si le résultat obtenu est identique lorsque cet ordre diffère. Le processus de fusion des tuiles considère les segments de part et d'autre des bordures communes et réapplique sur ces segments une procédure de segmentation. L'avantage de cette technique est qu'elle est généralisable à n'importe quel critère d'homogénéité et n'importe quelle heuristique de fusion. Cependant, comme nous l'avons vu dans la section précédente, le tuilage modifie les segments résultants dans chaque tuile. Cela signifie que la solution proposée accepte la présence de segments sous-optimaux à l'intérieur des tuiles. Par conséquent, elle ne garantit pas un résultat identique à celui obtenu sans tuilage.

Enfin dans [63], les auteurs ont proposé une solution exacte pour l'algorithme de segmentation Mean-Shift [34] en utilisant le tuilage. Dans un premier temps, les opérations

instables utiles pour optimiser le temps d'exécution de l'algorithme ont été supprimées dans la version classique du Mean-Shift. L'algorithme Biconnected Component, qui est utilisé dans la version classique pour assigner une étiquette unique aux pixels qui ont convergé vers le même mode, a été remplacé par le Connected Component [33]. Avec cette nouvelle version, les auteurs proposent de diviser l'image en tuiles et d'ajouter autour de chaque tuile une marge de stabilité qui est une couronne de pixels d'une certaine largeur. Cette marge dépend de la taille fixe de la fenêtre spatiale utilisée pour déplacer le pixel vers son mode ainsi que du nombre d'itérations à appliquer. Ces deux paramètres sont configurés par l'utilisateur avant la procédure. Toutes les tuiles avec leur marge de stabilité sont segmentées séparément. Enfin, les auteurs ont également défini une stratégie pour fusionner les tuiles après la segmentation en utilisant une table d'équivalence des étiquettes pour fusionner les segments de part et d'autre des bordures communes des tuiles adjacentes.

Assurer des résultats identiques lorsque l'image en entrée ne peut être stockée en mémoire s'avère être problématique pour les algorithmes de segmentation par fusion de régions. Nous venons de voir quelques solutions pour ce type d'algorithme, mais aucune d'entre-elles ne garantit des résultats identiques à ceux obtenus si l'image pouvait être segmentée sans tuilage. Cependant, une solution exacte a été trouvée pour l'algorithme du Mean-Shift en utilisant une marge de stabilité pour chaque tuile. Dans nos travaux, nous avons étendu cette approche pour les algorithmes par fusion de régions en proposant une solution générique et exacte qui ne dépend pas des critères d'homogénéité et des heuristiques de fusion.

4.4 GRM : un algorithme piloté par les données

Nous avons vu dans la section 4.2 que le tuilage a un impact sur les segments résultants. Dans cette section, une analyse des différentes étapes de l'algorithme va nous permettre de comprendre les raisons de cette modification engendrée par le tuilage. D'après le chapitre 3, la procédure de segmentation par fusion de régions peut-être modélisée par des opérations successives sur un graphe. Au début de la procédure, chaque noeud du graphe représente un segment contenant un pixel et a 4 ou 8 segments adjacents suivant le choix de la connectivité pour le voisinage. Durant les différentes étapes de la procédure, la structure du graphe est modifiée à cause de la fusion de certaines paires de segments. En effet, fusionner une paire de segments nécessite de transformer localement le voisinage des segments adjacents aux segments de cette paire. La fusion entraîne la modification de certaines arêtes du graphe et le regroupement de 2 noeuds pour n'en former qu'un seul. Pour le passage à l'échelle de l'algorithme GRM, nous choisissons l'heuristique LMBF et nous ajoutons la contrainte qu'un segment ne peut fusionner qu'une seule fois au maximum à chaque itération. Au cours d'une itération, tous les noeuds doivent être explorés pour déterminer s'ils doivent fusionner. Cela implique qu'une nouvelle itération dépend de l'état du graphe à l'issue de l'itération précédente. Basée sur ces caractéristiques, la procédure de fusion de régions peut-être catégorisée

comme un algorithme piloté par les données [75].

Diviser l'image en tuiles revient à considérer un graphe de segments par tuile. La structure initiale de ces graphes est différente de celle du graphe initial sans tuilage dans la mesure où les segments situés sur les bordures communes des tuiles adjacentes voient leur voisinage modifié. Après avoir appliqué une itération de la procédure de fusion, ces segments peuvent éventuellement fusionner avec des segments adjacents différents de ceux avec lesquels ils auraient fusionné si l'image n'était pas divisée en tuiles. Par conséquent, des segments supplémentaires voient leur voisinage modifié à l'issue de la première itération. Ce phénomène se propage ainsi vers des segments situés à l'intérieur de la tuile au fil des itérations. La figure 4.6 illustre un scénario qui peut se produire lorsqu'une opération de tuilage est appliquée. Nous allons utiliser la notion de segments contagieux introduite dans [64, 65]. La figure 4.6a représente le graphe initial de l'image. La figure 4.6b représente le graphe divisé en 4 sous-graphes représentant 4 tuiles. Les segments verts représentent les segments contagieux car leur voisinage a été modifié à cause du découpage. La figure 4.6c représente le graphe de l'image après avoir appliqué une itération. La figure 4.6d représente les graphes de chaque tuile après avoir appliqué une itération. Les segments rouges sont les segments qui sont différents comparés à ceux obtenus dans le graphe 4.6c à cause de l'absence de certains segments adjacents dans leur voisinage. Les segments verts sont les nouveaux segments contagieux qui voient leur voisinage modifié. L'étude de cet impact a été faite dans [76] où les auteurs expliquent que le tuilage induit des segments sous-optimaux à cause de l'absence de connaissance de certains segments adjacents qui sont situés dans d'autres tuiles. Dans la suite, nous allons délimiter les zones d'interaction des segments afin de proposer une marge de stabilité pour chaque tuile en fonction du nombre d'itérations. Mais avant cela, introduisons la notion de stabilité pour les algorithmes de segmentation.

4.5 Stabilité des algorithmes de segmentation

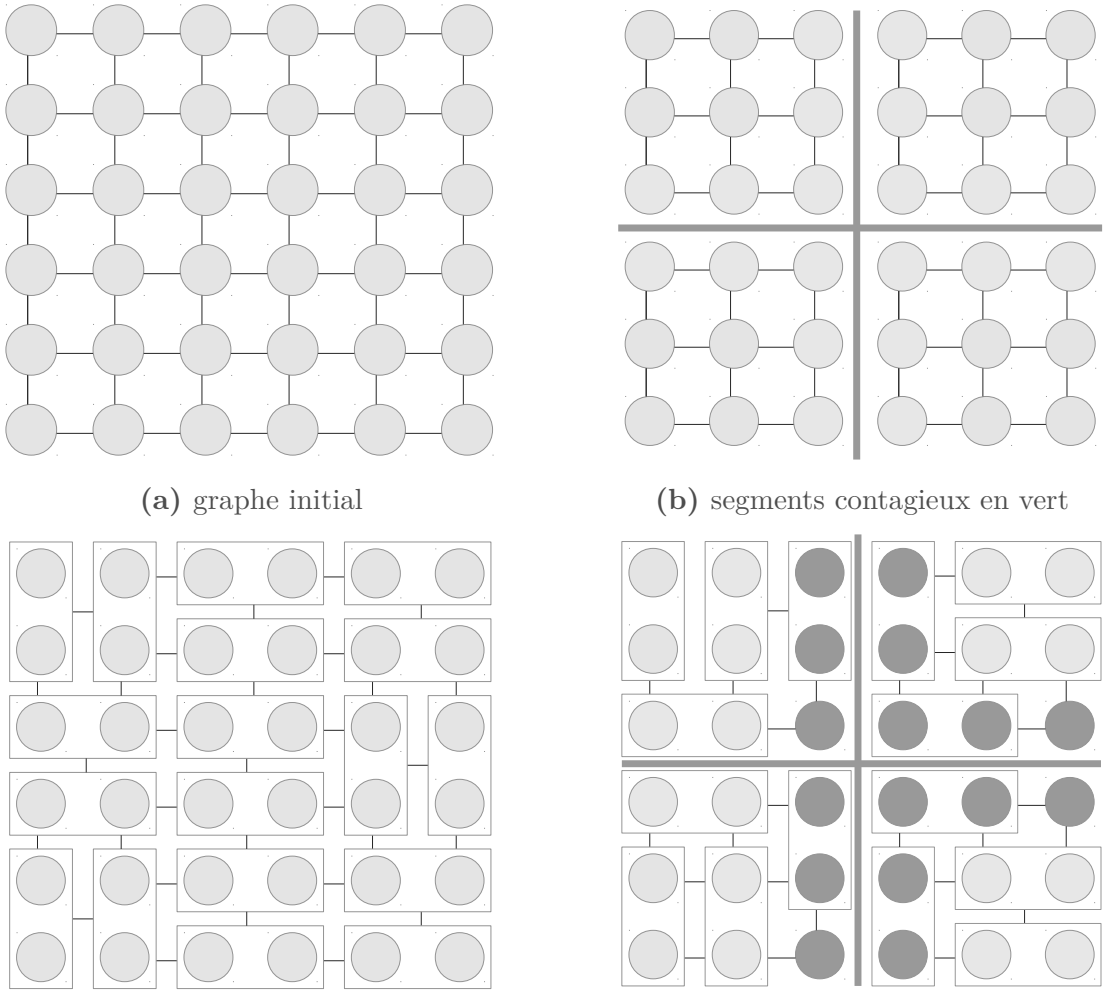
Nous rappelons que l'objectif de notre travail est d'assurer, en utilisant une stratégie de tuilage, des segments résultants identiques à ceux obtenus sans tuilage.

Soit I une image que nous désirons segmenter mais qui ne peut-être stockée en mémoire interne. $A(I)$ représente la partition de I en segments homogènes disjoints $(S_1, \ldots, S_n)$ où S_i est l'un des segments indexé par i. $T \subset I$ représente une tuile extraite de l'image I. Un segment indexé par j contenu dans la partition $A(T)$ est noté S'_j. Enfin, pour un segment $S \in A(I)$ et une tuile T, nous définissons $A_S(T)$ de la façon suivante :

$$A_S(T) = \{S' \in A(T) \setminus S' \subseteq S\} \tag{4.5}$$

$A_S(T)$ est donc l'ensemble des segments $S' \in A(T)$ qui sont totalement inclus dans le segment $S \in A(I)$. Avec ces notations, les auteurs dans [63] définissent une segmentation stable :

Definition Segmentation stable :
Un algorithme de segmentation A est stable si $\forall S \in A(I)$ et $\forall T \subset I$, les propriétés



(a) graphe initial

(b) segments contagieux en vert

(c) Après une itération sans tuilage

(d) Après une itération avec tuilage

Figure 4.6 – Impact du tuilage sur les segments au fil des itérations durant la procédure de segmentation par fusion de régions. La figure 4.6a représente le graphe initial de l'image. La figure 4.6b représente le graphe divisé en 4 sous-graphes représentant 4 tuiles. Les segments verts représentent les segments contagieux car leur voisinage a été modifié à cause du découpage. La figure 4.6c représente le graphe de l'image après avoir appliqué une itération. La figure 4.6d représente les graphes de chaque tuile après avoir appliqué une itération. Les segments rouges sont les segments qui sont différents comparés à ceux obtenus dans le graphe 4.6c à cause de l'absence de certains segments adjacents dans leur voisinage.

suivantes sont vérifiées :

$$S \subset T \Rightarrow \exists S' \in A(T) \setminus S = S' \tag{4.6}$$

$$S \cap T \neq \emptyset \Rightarrow S \cap T = \bigcup_{S' \in A_S(T)} S' \tag{4.7}$$

L'équation 4.6 représente la propriété de stabilité interne. En effet, si A est stable alors tout segment inclu totalement dans la tuile est identique à celui obtenu si l'image est segmentée sans tuilage. La propriété de stabilité interne correspond à la définition de la stabilité générique introduite dans la section 1.6.
L'équation 4.7 représente la propriété de couverture. Moins intuitive que la précédente, elle implique que tout segment dans $A(I)$ chevauchant les bordures communes des tuiles adjacentes est l'union de segments issus de la segmentation des tuiles. La figure 4.7 illustre la propriété de couverture.

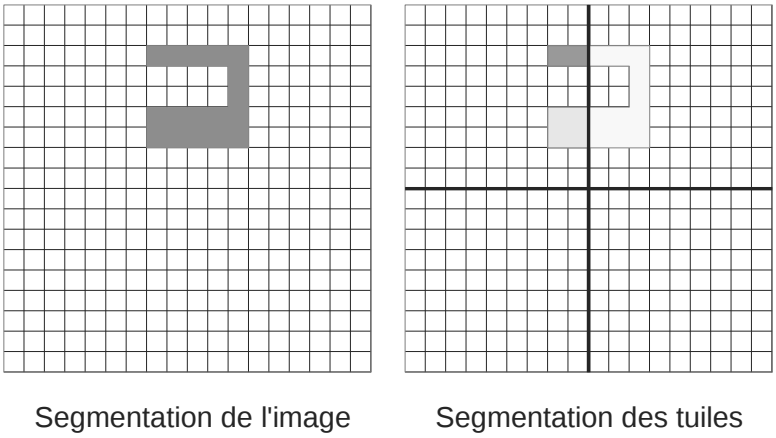


Figure 4.7 – La propriété de couverture. Les segments obtenus lors de la segmentation de l'image qui chevauchent les bordures communes des tuiles adjacentes sont l'union de segments obtenus lors de la segmentation des tuiles. Ceci est vrai à condition que l'algorithme de segmentation soit stable.

La propriéte de couverture introduite dans [6] diffère de la définition de la stabilité générique introduite dans 1.6 puisqu'il n'y a plus égalité des cardinaux des ensembles : $|A_{ref}(I)| < |A_{sca}(I)|$. Dans [6], une stratégie est ensuite utilisée pour fusionner tous les segments contenus de part et d'autre des bordures des tuiles pour former les segments chevauchant ces bordures. Après cette étape, nous retrouvons bel et bien la définition de la stabilité telle que nous l'avons introduite. Dans la suite de nos travaux, nous ne considérons pas la propriété de couverture et nous étendons la propriété de stabilité interne à l'ensemble des segments dans l'image grâce à la stratégie que nous proposons.

4.6 Zone d'influence d'un segment

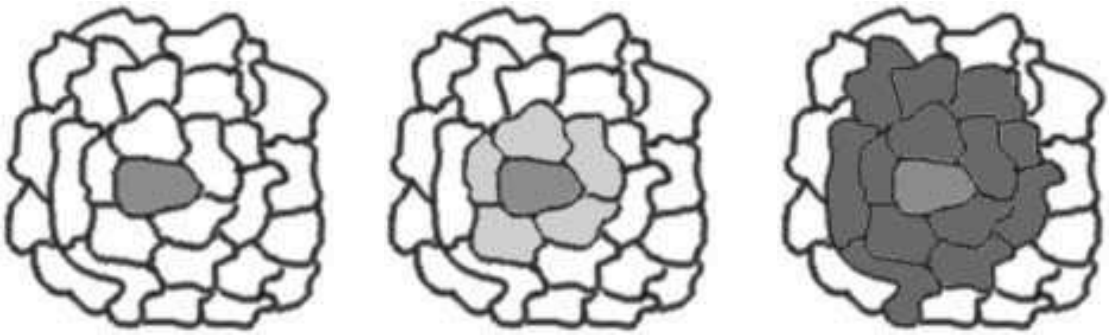
Nous considérons l'algorithme GRM et nous rappelons le principe de l'heuristique LMBF en supposant que le coût de fusion entre la paire de segments adjacents S_1 et S_2

est inférieur au seuil. S_1 fusionne avec S_2 si les conditions suivantes sont vérifiées :

1. S_2 est le segment adjacent pour lequel le coût de fusion est le minimum avec S_1 parmi les segments adjacents de S_1.

2. S_1 est le segment adjacent pour lequel le coût de fusion est le minimum avec S_2 parmi les segments adjacents de S_2.

Nous rappelons également que l'algorithme GRM peut être configuré de telle façon qu'un segment ne peut fusionner qu'une seule fois à chaque itération (voir la discussion dans la section 2.5). Par conséquent, à chaque itération, la zone d'interaction d'un segment S_i est limitée à la liste de ses segments adjacents. Puisque l'heuristique LMBF est utilisée, la liste des segments adjacents pour chaque segment adjacent à S_i est nécessaire pour vérifier la condition de mutualité. Pour résumer, la zone d'influence d'un segment correspond à l'union de ses segments adjacents et des segments adjacents à ses segments adjacents. Considérer cette zone d'influence pour chaque segment à chaque itération est une condition suffisante pour assurer la stabilité de l'algorithme GRM. La figure 4.8 représente la zone d'influence d'un segment pour effectuer une itération. De plus, nous remarquons que pour les autres heuristiques de fusion (F, BF et GMBF), la zone d'influence est plus petite. L'heuristique LMBF représente donc le cas le plus défavorable. La zone d'influence illustrée dans la figure 4.8 est donc générique à toutes les heuristiques de fusion.



(a) **(b)** **(c)**

Figure 4.8 – Zone d'influence d'un segment pour une itération de la procédure de segmentation par fusion de régions. La figure 4.8a représente un segment de l'image S_i. La figure 4.8b représente la liste des segments adjacents (en vert) nécessaire pour déterminer le segment adjacent le plus similaire à S_i. La figure 4.8c représente l'ensemble des segments adjacents aux segments adjacents de S_i (en bleu) nécessaire pour vérifier la condition de mutualité. Cet ensemble représente la zone d'influence de S_i.

L'objectif est d'étudier l'évolution de cette zone d'influence sur plusieurs itérations afin de proposer une marge de stabilité.

4.7 Calcul de la marge de stabilité

Connaissant la zone d'influence d'un segment pour une itération, l'objectif de cette section est de déterminer l'évolution de cette zone en fonction du nombre d'itérations appliquées avec l'algorithme GRM et de proposer une expression de la marge de stabilité. La valeur de la marge de stabilité représente la largeur d'une couronne de pixels située autour de la tuile. Le rôle de cette couronne de stabilité est d'assurer que les segments contenus dans la tuile soient identiques à ceux obtenus dans le cas où l'image pourrait être segmentée sans tuilage. La valeur de la marge de stabilité est notée M_n où n est le nombre d'itérations de la procédure de segmentation. Elle est mesurée en pixels et est illustrée par la figure 4.9.

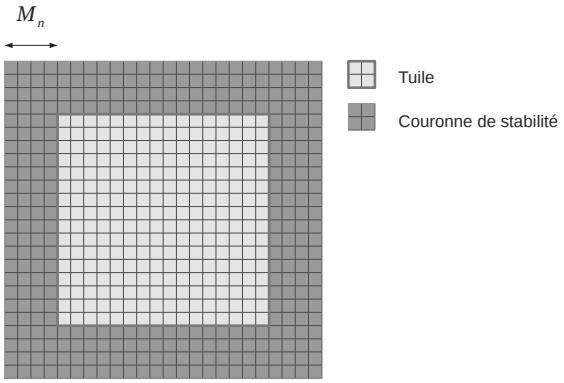
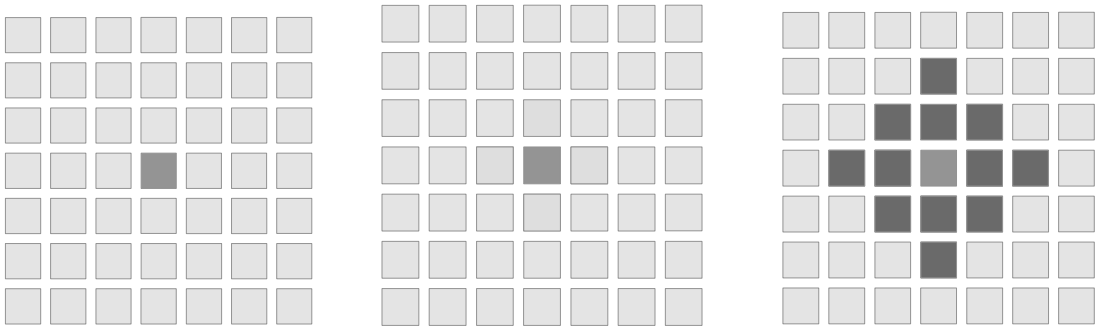


Figure 4.9 – Représentation de la marge de stabilité M_n pour une tuile. La valeur de cette marge dépend du nombre d'itérations que nous voulons appliquer avec l'algorithme GRM.

Au début de la procédure, chaque segment est un pixel et a, selon la connectivité, 4 ou 8 segments adjacents. Le type de connectivité n'ayant aucune influence sur l'expression finale de la marge de stabilité, nous choisissons un voisinage 4-connexe. Un segment initial a donc au départ 4 segments adjacents situés en haut, à droite, en bas et à gauche. Pour un segment S, nous notons l'ensemble de ses segments adjacents S_* et l'ensemble des segments adjacents aux segments adjacents S_{**}. Avec cette notation, le nombre d'étoiles $*$ qui indexent S représente le nombre de couches d'adjacence autour de S. Ainsi, S_{k*} est l'ensemble des segments adjacents aux segments contenus dans $S_{(k-1)*}$.

D'après la zone d'influence établie à la section précédente, il est nécessaire de considérer $S_* \cup S_{2*}$ pour chaque segment afin d'appliquer une itération stable. Pour effectuer la première itération, cela revient à considérer une couronne de largeur 2 pixels autour de chaque segment. Elle est illustrée dans la figure 4.10. Ainsi, une marge de stabilité de 2 pixels doit être considérée autour de chaque tuile pour effectuer la première itération ($M_1 = 2$).

La marge de stabilité peut être définie par récurrence. Supposons M_n la marge de stabilité nécessaire pour effectuer les n premières itérations et déterminons M_{n+1}. Soit



(a) S (b) S_* (c) S_{**}

Figure 4.10 – Zone d'influence d'un segment initial. La figure 4.10b représente S_* et la figure 4.10c représente $S_* \cup S_{**}$.

S un segment dans un certain état après avoir appliqué les n premières itérations. Pour appliquer une itération supplémentaire, l'ensemble des couches d'adjacence contenues dans $S_* \cup S_{2*}$ est nécessaire (Figure 4.8). Le nombre de pixels maximum contenus dans S après les n premières itérations est égal à 2^n puisqu'un segment ne peut fusionner qu'une seule fois à chaque itération. Nous notons, dans le cas où des heuristiques permettraient plus de fusions, que le calcul pourrait être ici adapté. La zone d'influence de S ne doit pas contenir de segments différents de ceux que nous aurions obtenus si l'image était segmentée sans tuilage. Par conséquent, une marge de stabilité doit être considérée pour chaque segment dans la zone d'influence de S pour les n premières itérations, c'est-à-dire M_n. De plus, nous considérons la pire situation où les segments croissent toujours suivant la même direction (segments filiformes). Avec ces hypothèses et d'après la Figure 4.11, M_{n+1} peut-être exprimée en fonction de M_n de la façon suivante :

$$M_{n+1} = 2^{n+1} + M_n, \text{ avec } M_0 = 0. \tag{4.8}$$

qui, après simplification, devient :

$$M_n = 2^{n+1} - 2 \tag{4.9}$$

Ainsi, lors d'une opération de tuilage, la première étape consiste à définir le nombre d'itérations à appliquer, noté n. La seconde étape consiste à diviser l'image en tuiles en considérant une marge de stabilité supplémentaire. Elle est représentée par une couronne de largeur M_n pixels autour de chaque tuile. Enfin, n itérations avec l'algorithme GRM peuvent être appliquées sur chaque tuile séparément. Il est important de noter que les pixels situés dans la couronne de stabilité sont aussi segmentés. Par conséquent, à l'issue de la segmentation, nous distinguons 2 types de segments résultants :

- Ceux contenant au moins 1 pixel dans la tuile. L'ensemble de ces segments est noté S_s et représente les segments stables.

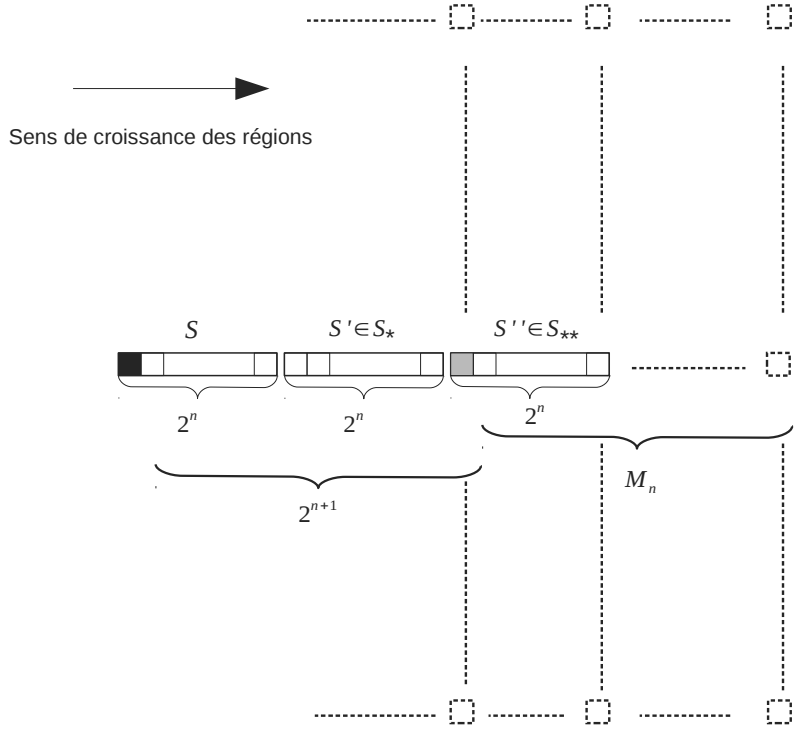


Figure 4.11 – Marge de stabilité , notée M_{n+1}, pour les $n+1$ premières itérations en fonction de M_n et de n.

- Ceux qui sont totalement inclus dans la couronne de stabilité. L'ensemble de ces segments est noté S_u et représente les segments instables.

Les segments contenus dans S_s sont identiques à ceux obtenus si l'image était segmentée sans tuilage puisque la marge de stabilité est considérée pour tous les pixels contenus dans la tuile. En revanche, les segments contenus dans S_u peuvent être potentiellement différents puisque la marge de stabilité n'est considérée pour aucun des pixels qu'ils contiennent. Ces segments sont ainsi instables et ne doivent pas être pris en compte par la suite.
Soit $TM \subset I$ une tuile avec sa marge de stabilité extraite de I et $T \subset TM$ la tuile sans sa marge de stabilité. $A(TM)$ est l'ensemble des segments résultants obtenus à l'issue de la segmentation de la tuile avec sa marge de stabilité. $A(I)$ est l'ensemble des segments résultants obtenus à l'issue de la segmentation de I sans tuilage. Soit S' un segment contenu dans $A(TM)$ et S est un segment contenu dans $A(I)$. Avec ces notations, nous définissons la stabilité de A de la façon suivante :

Definition Segmentation par fusion de régions stable :
Un algorithme de segmentation A est stable si $\forall S \in A(I)$ et $\forall TM \subset I$, la propriétée suivante est vérifiée :

$$S \cap T \neq \emptyset \Rightarrow \exists S' \in A(TM) \setminus S = S' \tag{4.10}$$

La stabilité est basée sur cette définition dans la suite de nos travaux. Le prochain chapitre décrit le nouvel algorithme échelonnable permettant de segmenter des images de taille arbitraire tout en garantissant des résultats identiques à ceux obtenus sans tuilage.

Chapitre 5

Algorithme échelonnable proposé : LSGRM

5.1 Aperçu général de la LSGRM

Pour rendre compréhensible la stratégie employée dans notre algorithme, baptisé algorithme LSGRM ("Large Scale Generic Region Merging"), nous décrivons dans cette section ses étapes principales sans rentrer dans les détails techniques. Leur explication plus détaillée est effectuée dans les sections suivantes. Notre explication de l'algorithme est basée sur le diagramme d'exécution de celui-ci illustré dans la figure 5.1. Chaque action dans ce diagramme est identifiée par un numéro unique.

Nous rappelons que nous considérons un ordinateur décrit dans le chapitre 1 section 1.2. La capacité de la mémoire interne disponible est notée M_i. Nous supposons que la quantité de mémoire externe est toujours suffisante pour stocker les données.

L'objectif est d'expliquer quelle est la stratégie d'exécution lorsque le graphe initial des segments de l'image ne peut être stocké en mémoire interne. M_g représente la quantité de mémoire nécessaire pour stocker ce graphe. L'algorithme LSGRM décompose la procédure de segmentation en une succession de segmentations partielles des tuiles de l'image.

La première étape (action 1 dans le diagramme 5.1) consiste à déterminer la taille des tuiles et la valeur de la marge de stabilité pour la première segmentation partielle sachant M_i. Vu le caractère exponentiel de la marge de stabilité en fonction du nombre d'itérations, il sera souvent impossible de segmenter les tuiles en une seule fois. Par conséquent, plusieurs segmentations partielles seront souvent nécessaires pour chaque tuile. Plusieurs stratégies sont possibles pour déterminer ces paramètres. Choisir une marge de stabilité élevée implique une réduction plus importante du nombre de segments dans chaque tuile et ainsi une diminution de la quantité de mémoire nécessaire pour stocker le graphe global de l'image. Cependant, 2 inconvénients sont à noter avec cette stratégie. Le premier inconvénient est que lorsque la valeur de la marge est élevée, le nombre de segments instables à traiter est important. Ceci augmente ainsi le temps

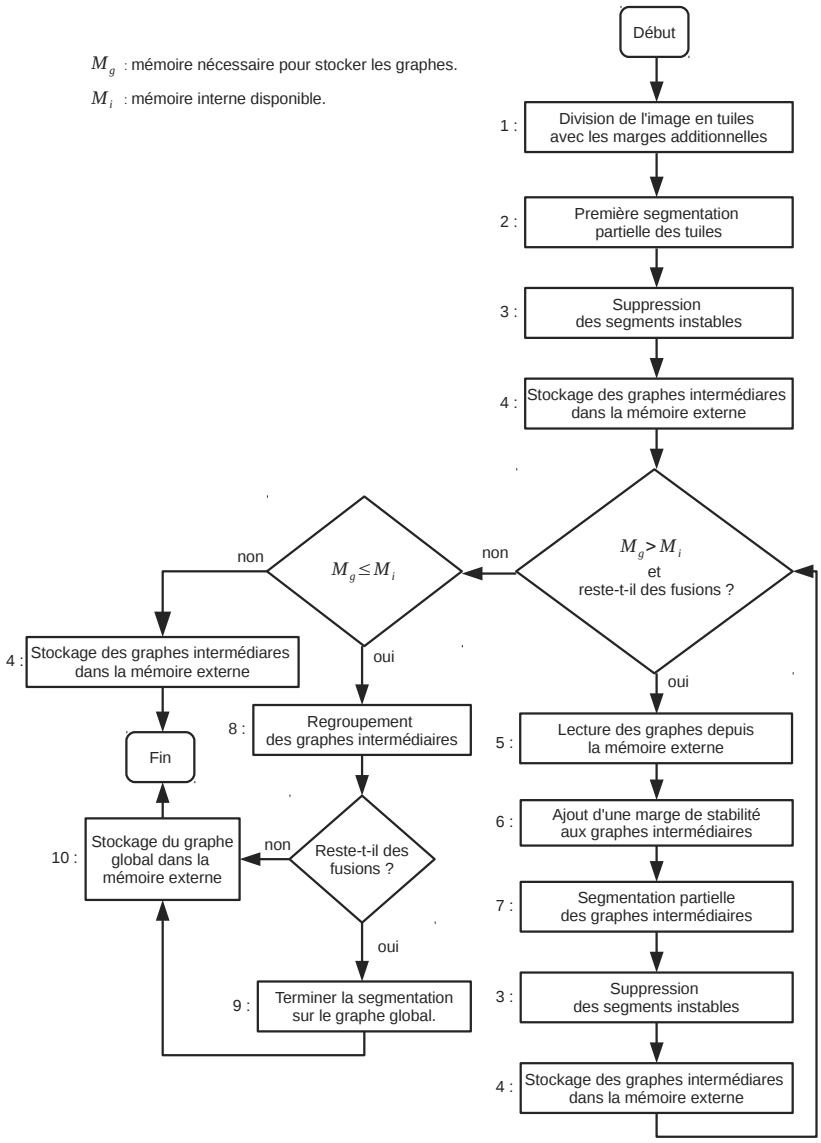


Figure 5.1 – Diagramme d'exécution de l'algorithme LSGRM

d'exécution inutilement. Le second inconvénient est que cela implique également une diminution de la taille des tuiles et par conséquent un temps de communication plus élevé dû aux nombreux accès en lecture et écriture dans la mémoire externe durant la procédure. A l'inverse, diminuer la valeur de la marge de stabilité au profit d'une taille des tuiles plus importante implique moins d'accès dans la mémoire externe. Par contre, cette stratégie induit plus de segmentations partielles car la vitesse de diminution de la quantité de mémoire nécessaire pour stocker le graphe global de l'image est plus faible. Un compromis entre la taille des tuiles et la valeur de la marge de stabilité doit être déterminé. Le développement d'une heuristique pour trouver le compromis optimal n'a

pas été traité dans cette thèse mais devrait être investigué dans le futur. Une fois la taille des tuiles et la marge de stabilité déterminées, l'image est divisée en tuiles avec leur couronne de stabilité supplémentaire.

Supposons M_{n_1} la valeur de la marge de stabilité pour la première segmentation partielle. La seconde étape (action 2 dans le diagramme 5.1) consiste à segmenter chaque tuile avec sa couronne de stabilité séparément en appliquant n_1 itérations. A l'issue de la segmentation partielle d'une tuile avec sa couronne de stabilité, un graphe intermédiaire de segments est obtenu. Ce graphe contient à la fois des segments stables (S_s) et des segments instables (S_u) comme nous l'avons expliqué dans la section 4.7. Seuls les segments stables doivent être conservés dans le graphe. Une étape de suppression des segments instables est nécessaire à l'issue de la segmentation partielle de chaque tuile (action 3 dans le diagramme 5.1) . Après avoir supprimé les segments instables, chaque graphe intermédiaire est stocké dans la mémoire externe (action 4 dans le diagramme 5.1). L'ensemble de cette étape constitue la première segmentation partielle des tuiles.

La troisième étape est une boucle qui s'arrête lorsque nous détectons que le graphe global de l'image peut être stocké en mémoire interne, c'est-à-dire lorsque $M_g < M_i$ ou lorsqu'aucune fusion n'a été appliquée dans chacune des tuiles durant la segmentation partielle précédente. Le corps de la boucle consiste à lire les graphes depuis la mémoire externe (action 5 dans le diagramme 5.1) puis à effectuer les opérations décrites lors de la seconde étape à la différence que la couronne de stabilité doit être ajoutée à un graphe et non pas à une image (action 6 dans le diagramme 5.1). Chaque noeud du graphe intermédiaire n'est plus un pixel mais un segment qui possède un certain nombre d'arêtes. Dans cette situation, la marge de stabilité n'est plus exprimée en pixels mais en segments et nous verrons par la suite comment étendre l'expression de la marge de stabilité dans ce nouveau contexte.

Lorsque la boucle s'arrête, il y a 2 possibilités. La première possibilité est que le graphe global de l'image soit stocké en mémoire interne. Dans ce cas, les graphes intermédiaires sont fusionnés pour constituer le graphe global de l'image (action 8 dans le diagramme 5.1). Si la segmentation n'est pas terminée, alors elle est achevée en appliquant les dernières itérations sur le graphe de l'image (action 9 dans le diagramme 5.1) puis le graphe résultant est stocké dans la mémoire externe (action 10 dans le diagramme 5.1). La seconde possibilité est qu'aucune fusion n'ait été détectée dans chaque graphe intermédiaire lors de la dernière segmentation partielle dans la boucle. Dans ce cas, les graphes de chaque tuile représentent les partitions finales et sont stockés dans la mémoire externe (action 4 dans le diagramme 5.1). Nous notons une propriété intéressante de notre solution qui ne requiert pas que le graphe résultant de l'image puisse être stocké en mémoire interne. En effet, la segmentation peut être achevée sur chaque tuile de l'image indépendamment.

Dans la suite de ce chapitre, les opérations décrites brièvement dans cette section sont détaillées et leurs complexités en temps et en espace analysées. Nous ne détaillerons pas l'opération qui consiste à segmenter un graphe car son principe de fonctionnement et sa complexité ont déjà été expliqués dans le chapitre 3.

5.2 Suppression des segments instables

Cette étape correspond à l'action 3 dans le diagramme 5.1. Soit une tuile avec sa couronne de stabilité, notée TM, et T la tuile sans la couronne de stabilité. A l'issue d'une segmentation partielle de TM, un graphe intermédiaire est obtenu. Ce graphe contient à la fois des segments stables et des segments instables. Nous rappelons qu'un segment S est instable si $S \cap T \neq \emptyset$. L'objectif est donc de supprimer du graphe tous les segments S' totalement inclus dans la couronne de stabilité : $S' \subset (TM \setminus T)$. Une tuile est une portion rectangulaire de l'image qui peut être délimitée par les coordonnées du coin supérieur gauche (x_{min}, y_{min}) et les coordonnées du coin inférieur droit (x_{max}, y_{max}). Les limites d'une tuile sont illustrées par la figure 5.2.

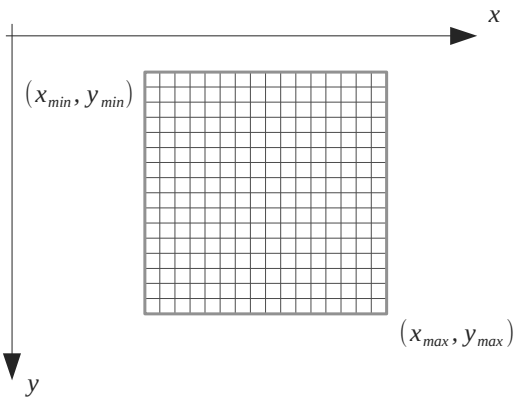


Figure 5.2 – Délimitation d'une tuile dans l'image.

L'idée consiste à parcourir les segments du graphe et à détecter si le segment contient au moins 1 pixel à l'intérieur de la tuile. Pour chaque segment, nous avons à disposition son contour et nous avons vu dans le chapitre 3 section 3.2.4 que nous pouvons, à partir du contour, générer les pixels internes. Cependant, la complexité de cette opération est linéaire suivant le nombre de déplacements le long du contour de segments et cela doit être effectué pour chaque segment dans le graphe. Si N est le nombre de segments dans le graphe et $\bar{M}$ le nombre de déplacements maximum par segment alors la complexité en temps est $O(N \times \bar{M})$. Même si cette complexité en temps est acceptable nous pouvons faire mieux. En effet, pour chaque segment nous avons également à disposition sa boîte englobante qui est illustrée par la figure 3.3 dans le chapitre 3 section 3.2.2. Cette boîte est caractérisée par les coordonnées du coin supérieur gauche (x, y) ainsi que par sa largeur w et sa hauteur h. Sachant que la plupart des segments sont situés soit à l'intérieur de la tuile soit à l'extérieur, nous pouvons éliminer une bonne partie des segments pour lesquels il n'est pas nécessaire de générer les pixels.

En effet si la condition suivante est vérifiée :

$$\begin{cases} x & \geq x_{min} \text{ et} \\ y & \geq y_{min} \text{ et} \\ x+w & \leq x_{max} \text{ et} \\ y+h & \leq y_{max} \end{cases} \tag{5.1}$$

alors le segment est totalement inclus dans la tuile et est donc stable. A l'inverse, si la condition suivante est vérifiée :

$$\begin{cases} x & > x_{max} \text{ ou} \\ y & > y_{max} \text{ ou} \\ x+w & < x_{min} \text{ ou} \\ y+h & < y_{min} \end{cases} \tag{5.2}$$

alors le segment est instable et sera supprimé du graphe. En revanche, pour les segments dont la boîte englobante chevauche la bordure de la tuile, aucune décision ne peut être prise. Dans cette situation, la seule solution est de générer les pixels pour détecter si au moins l'un d'eux est situé à l'intérieur de la tuile. Si c'est le cas, le segment est stable, sinon il est instable et supprimé du graphe. Si nous connaissons la taille moyenne des boîtes englobantes, nous pouvons calculer le ratio de segments pour lesquels il faut générer des pixels. Soit $\bar{N}$ le nombre maximum de segments pour lesquels il faut générer les pixels, $\bar{M}$ le nombre de déplacements maximum pour un segment et $\bar{P}$ le nombre de pixels maximum dans un segment. La complexité en temps devient $O(N + \bar{N} \times \bar{M})$ et la complexité en espace est $O(\bar{P})$ car il faut stocker la liste des pixels générée pour chaque segment.

5.3 Stockage et chargement des graphes

5.3.1 Stockage des graphes dans la mémoire externe

Cette étape correspond à l'action 4 dans le diagramme 5.1. Une fois les segments instables supprimés, l'opération suivante consiste à écrire le graphe dans la mémoire externe. Pour cela, 2 fichiers sont écrits. Le premier fichier, noté F_1, contient, pour chaque segment, le contour, la boîte englobante ainsi que les attributs spécifiques au critère d'homogénéité :

$$\begin{bmatrix} id_1 \rightarrow [bbox_1, contour_1, \ldots] \\ . \\ . \\ . \\ id_N \rightarrow [bbox_N, contour_N, \ldots] \end{bmatrix}$$

où id_i, $bbox_i$ et $contour_i$ sont respectivement l'identifiant, la boîte englobante et le contour du segment S_i indexé par i dans le graphe.

Le second fichier, noté F_2, contient, pour chaque segment, son voisinage :

$$\begin{bmatrix} id_1 \rightarrow [id_{1_1}, \ldots, id_{1_{v1}}] \\ . \\ . \\ . \\ id_N \rightarrow [id_{N_1}, \ldots, id_{N_{vN}}] \end{bmatrix}$$

où id_i est l'identifiant du segment S_i et id_{i_j} pour $j \in [1, vi]$ est l'identifiant d'un segment adjacent à S_i.
La complexité en temps de cette opération est $O(N \times (\bar{E} + \bar{M})$. La complexité en espace est $O(N + E + M)$.

5.3.2 Chargement des graphes depuis la mémoire externe

Cette étape correspond à l'action 5 dans le diagramme 5.1. Le chargement et la reconstruction du graphe s'effectuent en 2 étapes. Les noeuds du graphe sont d'abord construits à partir de la lecture de F_1. Une table de hachage est utilisée pour associer à chaque identifiant un pointeur vers le noeud correspondant dans le graphe. La seconde étape consiste à construire les arêtes du graphe en utilisant la table de hachage qui permet d'accéder à un noeud dans le graphe en temps constant. Malgré le fait qu'elle nécessite un espace mémoire supplémentaire de complexité $O(N)$, elle permet d'éviter la recherche des segments adjacents dans le graphe pour chaque segment, ce qui induirait une complexité en temps quadratique $O(N^2 \times \bar{E})$. Au final, la complexité en temps de l'opération est $O(N \times (\bar{E} + \bar{M}))$ et la complexité en espace est $O(2N + M + E)$.

5.4 Ajout d'une marge de stabilité à un graphe

Cette étape correspond à l'action 6 dans le diagramme 5.1. Nous sommes dans la situation où une segmentation partielle doit être appliquée sur un graphe de segments d'une tuile. Soit n le nombre d'itérations que nous voulons appliquer. Pour garantir la stabilité, il est donc nécessaire de considérer une couronne de stabilité autour du graphe. Cependant, la largeur de cette couronne ne doit plus être égale à M_n pixels. En effet, le graphe n'est plus composé de pixels mais de segments contenant des ensembles de pixels connectés. De plus, ces segments peuvent avoir une forme et un voisinage quelconque.
Par conséquent, la marge de stabilité ne doit plus être mesurée en pixels mais en segments. Si nous supposons que $n = 1$, alors la couronne de stabilité à considérer autour de chaque segment S du graphe est $S_* \cup S_{**}$ (figure 4.8). Nous rappelons que S_{i*} désigne la i^{ème} couche d'adjacence du segment S. Lorsque $i = 1$, alors $S_{1*} \equiv S_*$ représente les segments adjacents de S. $S_{2*} \equiv S_{**}$ représente les segments adjacents de tous les segments dans S_* et constitue la seconde couche d'adjacence de S. Nous pouvons ainsi faire l'analogie entre la marge exprimée en pixels et la marge exprimée en segments. En effet, pour effectuer les n premières itérations, nous avons déterminé que la marge de stabilité

vaut $2^{n+1} - 2$ (équation 4.9). Par analogie, la couronne de stabilité à considérer autour de chaque segment S dans le graphe pour effectuer n itérations supplémentaires est l'ensemble des segments contenus dans $\bigcup_{i=1}^{2^{n+1}-2} S_{i*}$. Par exemple, la couronne de stabilité nécessaire autour de chaque segment lorsque $n = 2$ est l'ensemble des segments contenus dans $\bigcup_{i=1}^{6} S_{i*}$.

La plupart des segments contenus dans la marge sont déjà dans le graphe. En revanche, pour les segments proches des bordures de la tuile, certains des segments contenus dans leur couronne de stabilité sont situés dans les graphes de segments des tuiles adjacentes et sont manquants. Il doivent donc être récupérés. La figure 5.3 illustre l'ajout de la couronne de stabilité aux segments du graphe dans le cas où $n = 1$. Pour considérer la couronne de stabilité autour de chaque segment du graphe, il est seulement nécessaire de la considérer pour les segments situés en bordure du graphe (segments rouges dans la figure 11.2b). Par contre, cela implique d'aller chercher des segments dans les graphes des tuiles adjacentes (figure 5.3c).

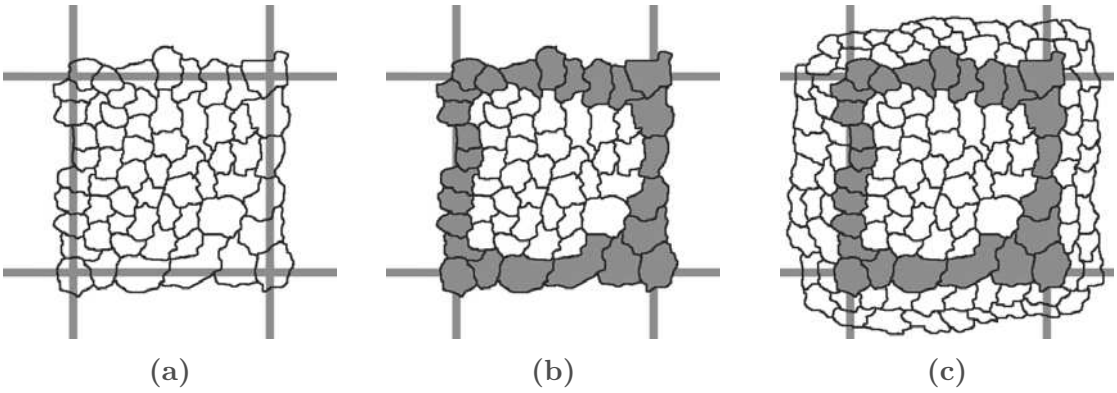


Figure 5.3 – Ajout de la couronne de stabilité dans le cas où $n = 1$. Il est nécessaire de considérer l'ensemble des segments contenus dans $\bigcup_{i=1}^{2} S_{i*}$ autour de chaque segment S du graphe.

Récupérer les segments adjacents manquants dans les graphes des tuiles adjacentes s'effectue de la façon suivante. Soit une bordure commune entre deux tuiles adjacentes. Notons (z_b) les coordonnées de la bordure interne de la tuile courante et (z'_b) les coordonnées de la bordure interne de la tuile adjacente. La première étape consiste à détecter les segments situés sur la bordure commune dans le graphe de la tuile adjacente. Cela consiste à explorer le graphe des segments de la tuile adjacente et à détecter les segments qui contiennent au moins un pixel dont les coordonnées correspondent à la bordure. Ces segments sont stockés dans une liste doublement chaînée P_b. Si N est le nombre de segments dans le graphe et N_b le nombre de segments en bordure du graphe alors la complexité en espace est $O(N + N_b)$ et la complexité en temps $O(N)$.

Une fois les segments en bordure de la tuile détectés, l'étape suivante consiste à extraire pour chaque segment en bordure, noté S^b, l'ensemble $\bigcup_{i=1}^{2^{n+1}-2} S^b_{i*}$ dans le graphe. Pour effectuer cela, nous allons utiliser l'algorithme de parcours en profondeur d'un

graphe décrit dans [11] Chapitre 22 Section 3. Cet algorithme consiste à effectuer une recherche en profondeur dans un graphe en explorant toujours les arêtes du dernier noeud visité. Dans notre situation, il y a une profondeur limite d'exploration qui correspond au nombre de couches d'adjacence nécessaires autour de chaque segment S^b qui est égal à $2^{n+1} - 2$. Par conséquent, une table de hachage, notée C_b, est utilisée pour stocker les segments explorés où la clé est un pointeur vers un segment et la valeur la profondeur de visite du segment. Dans un premier temps, les segments S^b sont stockés dans C_b et leur profondeur est égale à 0. Lors de la recherche dans le graphe, l'exploration d'un nouveau segment S à la profondeur p avec $p < 2^{n+1} - 2$ s'effectue de la façon suivante :

1. Nous vérifions si le segment est déjà dans C_b.

2. Si c'est le cas :

   (a) Si la profondeur dans C_b est strictement inférieure à p alors nous arrêtons car les couches d'adjacence pour ce segment ont déjà été explorées.

   (b) Sinon, nous mettons à jour C_b avec la nouvelle valeur p et nous explorons ses couches d'adjacence.

3. Si ce n'est pas le cas, alors nous insérons la paire (S, p) dans C_b et nous explorons ses couches d'adjacence.

Le pseudo algorithme pour extraire les couches d'adjacence est illustré par l'algorithme 5.1 où p_{max} représente la profondeur maximum d'exploration et $s.aretes$ représente la liste des segments adjacents du segment s.

Analysons la complexité en espace de l'algorithme 5.1 en supposant que N est le nombre de segments dans le graphe, E le nombre d'arrêtes et M le nombre de déplacements le long des contours. N_b est le nombre de segments situés sur la bordure du graphe et N_s est le nombre total de segments dans les couches d'adjacence des segments situés sur la bordure. Le graphe des segments doit être stocké en mémoire, c'est-à-dire N segments, E arêtes et M déplacements. P_b est une liste chaînée contenant N_b pointeurs de segment et C_b est une table de hachage contenant N_s paires composées d'un pointeur de segment et de la valeur de la profondeur. La complexité en espace est alors $O(N + N_b + N_s + M + E)$. Ainsi, choisir le nombre d'itérations à appliquer dépend de la quantité de mémoire nécessaire pour effectuer cette opération. Il faut être capable de stocker le graphe plus les couches d'adjacence des segments en bordure du graphe.

Analysons la complexité en temps de cette opération. Initialiser C_b avec les segments en bordure du graphe requiert N_b opérations. Ensuite, pour chaque segment en bordure, nous appelons la fonction Extract. Lors de la récursion, la fonction Extract est appelée pour chaque nouveau segment à explorer. Au total, elle est appelée $N_b + N_s$ fois. Les opérations effectuées dans la fonction Extract sont des opérations d'accès et de mise à jour de la table de hachage. Ces opérations s'effectuent en moyenne en temps constant. Par conséquent, la complexité en temps de l'étape d'extraction est $O(N_b + N_s)$.

```
1: procédure ExploreGrapheAdjacent(P_b, C_b, p_max)
2:     pour chaque s ∈ P_b exécute
3:         C_b[s] = 0
4:     fin pour
5:     pour chaque s ∈ P_b exécute
6:         Extract(s, 0, C_b, p_max)
7:     fin pour
8: fin procédure
```

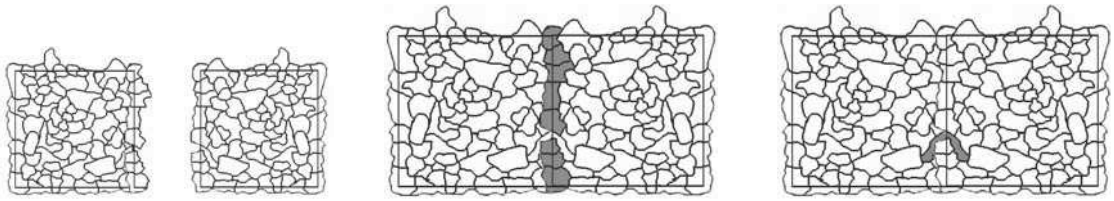
```
1: procédure Extract(s, p, C_b, p_max)
2:     si p > p_max alors
3:         fin.
4:     sinon
5:         si s ∈ C_b alors
6:             si p ≤ C_b[s] alors
7:                 C_b[s] = p
8:                 pour chaque s' ∈ s.aretes exécute
9:                     Extract(s', p + 1, C_b, p_max)
10:                fin pour
11:            sinon
12:                fin.
13:            fin si
14:        sinon
15:            Insère (s, p) dans C_b.
16:            pour chaque s' ∈ s.aretes exécute
17:                Extract(s', p + 1, C_b, p_max)
18:            fin pour
19:        fin si
20:    fin si
21: fin procédure
```

Algorithme 5.1 – Extraction des couches d'adjacence pour des segments en bordure du graphe d'une tuile adjacente.

5.5 Fusion de graphes de segment

Cette opération est effectuée dans 2 situations. La première situation est lorsque nous devons fusionner les marges de stabilité extraites des graphes des tuiles adjacentes au graphe courant (à la fin de l'action 6 dans le diagramme 5.1). L'extraction des marges de stabilité a été décrite dans la section précédente. La seconde situation est lorsque nous avons détecté que le graphe global de l'image peut être stocké en mémoire interne. Dans cette situation, les graphes des tuiles de l'image sont alors fusionnés (action 8).

La figure 5.4 illustre le regroupement de 2 graphes. La première étape consiste à détecter les segments chevauchant la bordure commune entre les 2 graphes. Ces segments sont dupliqués car ils sont présents dans les 2 graphes. Ils sont identifiés en rouge dans la figure 5.4b et représentent un bon indicateur de la validité de l'algorithme LSGRM. En effet, si un segment chevauchant la bordure commune n'est pas dupliqué, cela signifie que notre algorithme n'est pas stable. Après avoir traité les segments dupliqués, il faut aussi considérer les segments situés de part et d'autre de la bordure commune. Ces segments sont identifiés en vert dans la figure 5.4c. En effet, des liens d'adjacence sont manquants entre ces segments et il est nécessaire de les ajouter.



(a) **(b)** **(c)**

Figure 5.4 – Fusion de 2 graphes pour former un seul graphe. La fusion est composée de 2 étapes : le traitement des segments dupliqués (en rouge) et le traitement des segments situés de part et d'autre de la bordure commune (en vert).

5.5.1 Traitement des segments dupliqués

La première étape consiste à détecter les segments dupliqués. La manière de procéder pour effectuer cette opération a déjà été expliquée dans la section précédente. Il suffit de parcourir les segments du graphe et de détecter si au moins un pixel est situé de part et d'autre de la bordure commune. Cependant, pour détecter les segments dupliqués nous allons ajouter des opérations supplémentaires. Une table de hachage est utilisée pour associer aux coordonnées de chaque pixel situé de par et d'autre de la bordure la liste des segments contenant ce pixel. De cette manière, si pour un pixel nous avons plus d'un segment dans la liste alors ces segments sont dupliqués. Par exemple, en analysant la figure 5.5, la table de hachage résultante est de la forme

$$\begin{bmatrix} 1 \rightarrow & S_1, S_1' \\ 2 \rightarrow & S_1, S_1' \\ . & \\ . & \\ . & \\ 11 \rightarrow & S_3 \\ 12 \rightarrow & S_2' \\ . & \\ . & \\ 24 \rightarrow & S_5, S_4' \end{bmatrix}$$

Ainsi, si L est la longueur de la bordure, alors il est nécessaire de stocker $2L$ entrées dans la table de hachage. La complexité mémoire de cette opération est $O(2L+2N+2M+2E)$ car il faut aussi stocker les 2 graphes en mémoire interne. La complexité en temps est $O(2N + 2N_b \times (\bar{P} - 1))$ où $\bar{P}$ est le nombre de pixels maximum par segment et N_b le nombre de segments situés sur la bordure de la tuile.

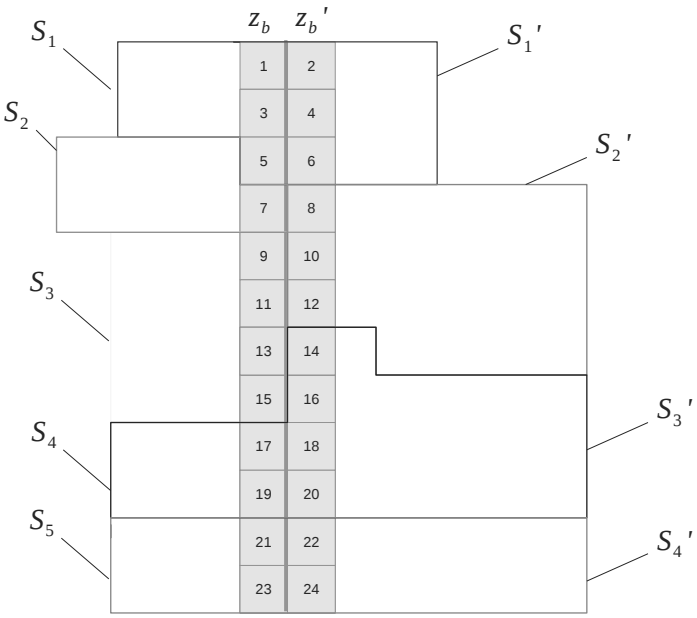


Figure 5.5 – Recherche des segments dupliqués.

Les segments qui chevauchent la bordure commune entre les 2 graphes sont dupliqués (segments rouges dans la figure 5.4b). Soit S_1 un segment chevauchant la bordure et S_1' le segment correspondant dans le graphe adjacent. Ces deux segments sont illustrés dans la figure 5.6. L'objectif est d'ajouter les arêtes de S_1' à S_1 si les segments adjacents ne sont pas déjà contenus dans le voisinage de S_1. Pour ce faire, nous parcourons les arêtes de S_1' et pour chaque segment adjacent, nous cherchons l'arête pointant vers S_1' et nous la supprimons. Nous cherchons également s'il existe une arête pointant vers S_1. Si ce n'est pas le cas alors nous rajoutons une arête entre S_1 et ce segment adjacent. La complexité en espace est constante $O(1)$. La complexité en temps est $O(\bar{E} \times \bar{E})$.

5.5.2 Mise à jour des arêtes

La dernière étape consiste à traiter les pixels des segments situés de part et d'autre de la bordure. C'est le cas pour les segments S_2, S_2' et S_3 dans la figure 5.5. Pour chaque pixel contenu dans la table de hachage et qui appartient à l'un de ces segments, nous calculons son voisinage 4-connexe. Si dans son voisinage, un pixel appartient à un segment qui n'est pas dans sa liste des segments adjacents, alors une nouvelle arête est construite.

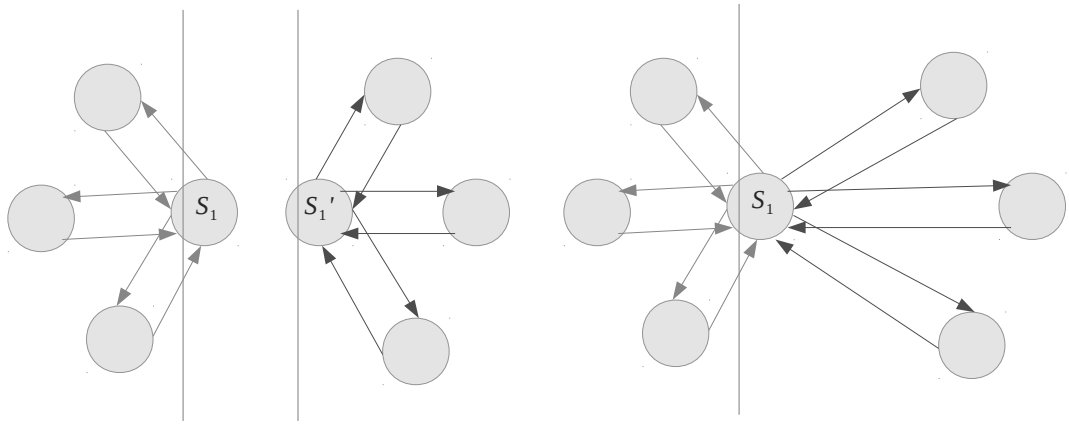


Figure 5.6 – Traitement d'un segment dupliqué

Il est aussi nécessaire de calculer la longueur de la frontière avec ce nouveau segment adjacent. Pour effectuer cela, nous examinons les pixels contenus dans le segment qui sont aussi présents dans la table de hachage. Pour chacun de ces pixels, nous déterminons son voisinage 4-connexe et nous comptons le nombre de pixels appartenant au nouveau segment adjacent pour incrémenter la longueur de la frontière commune. Par exemple, si nous considérons le segment S_3 dans la figure 5.5, en calculant le voisinage du pixel 9 nous détectons que le pixel 10 appartient au segment S_2'. Or, S_2' n'appartient pas à la liste des segments adjacents de S_3. Une arête est donc créée entre S_2' et S_3. Ensuite, nous parcourons les pixels contenus à la fois dans S_3 et dans la table de hachage pour déterminer la longueur de la frontière commune entre S_3 et S_2'. Ils sont 4 : les pixels 9, 11, 13 et 15. Pour chacun de ces pixels nous analysons leurs pixels voisins et comptons le nombre de fois qu'un pixel voisin appartient à S_2'. Dans notre cas, 2 fois car 10 est un pixel voisin de 9 et 12 est un pixel voisin de 11. La longueur de la frontière entre S_3 et S_2' vaut ainsi 2. Si P est le nombre de pixels dans la table de hachage et $\bar{E}$ le nombre d'arêtes maximum par segment alors la complexité en temps est $O(P \times \bar{E})$. La complexité en espace est $O(P)$ car il faut stocker les pixels ainsi que les pointeurs vers les segments dans lesquels ils sont contenus.

5.6 Example quantitatif des complexités de l'algorithme LSGRM

Pour illustrer les complexités en temps et en espace de l'algorithme LSGRM, nous avons réalisé la segmentation d'un extrait d'une scène Pléiades prise dans le Colorado aux États-Unis. La taille de l'extrait est 6000 × 6000 pixels. Pour l'experience, nous avons simulé un ordinateur avec une quantité de mémoire interne égale à 1 Go. La procédure a été réalisée séquentiellement afin de surveiller le processus et d'effectuer

les mesures. L'algorithme LSGRM utilise le critère d'homogénéité de Baatz & Schäpe avec les paramètres spécifiques suivants : $w_c = 0.7$, $w_s = 0.3$ et $scale = 60$. La taille d'un noeud du graphe en mémoire est 232 octets, celle d'une arête 24 octets et celle d'un déplacement le long du contour des segments 2 bits. Sachant qu'au départ un pixel représente un noeud et que chaque segment initial a 4 arêtes et 4 déplacements le long de son contour, le graphe initial des segments de l'image requiert un peu plus de 11 Go pour être stocké en mémoire interne.
Nous avons choisi de découper l'image en 36 tuiles de taille 1000×1000 pixels. Pour chaque tuile, nous avons choisi une marge de stabilité égale à 126 pixels. Cela correspond à 6 itérations lors de la première segmentation partielle des tuiles. Pour les segmentations partielles suivantes, nous avons choisi une marge de stabilité égale à $\bigcup_{i=1}^{14} S_{i*}$ permettant d'effectuer 3 itérations. Au total, 2 segmentations partielles ont été suffisantes pour pouvoir stocker le graphe des segments de l'image dans la mémoire interne. Le tableau 5.1 récapitule le temps d'exécution et la quantité de mémoire occupée par les graphes à l'issue des 3 grandes étapes, à savoir la première segmentation partielle, la seconde segmentation partielle et la segmentation finale.

Étape	Temps d'exécution	Mémoire occupée par le graphe
Première segmentation partielle	0h29	1,7 Go
Seconde segmentation partielle	0h22	924 Mo
Segmentation finale	0h20	180 Mo

Table 5.1 – Évolution du temps d'exécution et de la mémoire occupée par le graphe des segments au fil des étapes de l'algorithme LSGRM.

Le tableau 5.2 récapitule le temps d'exécution moyen pour chaque sous-étape lors de la première segmentation partielle des tuiles. Le temps d'exécution moyen est celui pour traiter une tuile.

Étape	Temps d'exécution
Segmentation	40 secondes
Suppression des segments instables	moins d'une seconde
Écriture du graphe en mémoire externe	moins d'une seconde
Extraction de la marge de stabilité	moins d'une seconde
Écriture de la marge de stabilité	moins d'une seconde

Table 5.2 – Répartition du temps d'exécution lors de la première segmentation partielle.

Le tableau 5.3 récapitule le temps d'exécution moyen pour chaque sous-étape lors de la seconde segmentation partielle des tuiles.

Enfin, le tableau 5.4 récapitule la répartition du temps d'exécution pour chaque sous-étape lors de la segmentation finale.

Étape	Temps d'exécution
Chargement du graphe	1 seconde
Ajout de la marge de stabilité	30 secondes
Segmentation	5 secondes
Suppression des segments instables	moins d'une seconde
Écriture du graphe en mémoire externe	moins d'une seconde

Table 5.3 – Répartition du temps d'exécution lors de la seconde segmentation partielle.

Étape	Temps d'exécution
Regroupement des graphes	40 secondes
Suppression des segments dupliqués et mise à jour des arêtes	0h13
Segmentation	0h06
Écriture du graphe	2 secondes

Table 5.4 – Répartition du temps d'exécution lors de la segmentation finale.

D'après cette étude, nous relevons des propriétés intéressantes de l'algorithme LS-GRM. La vitesse de réduction du nombre de segments est élevée lors des premières itérations, ce qui permet de stocker rapidement le graphe des segments de l'image en mémoire interne. Pour appuyer cette analyse, nous pouvons faire référence à l'étude faite dans la section 3.3.6. La segmentation du graphe et l'ajout de la marge de stabilité sont les 2 étapes qui prennent le plus de temps lors du traitement. Les écritures et lectures dans la mémoire externe sont négligeables.
Il est intéressant de constater que ces étapes sont facilement parallélisables sur plusieurs noeuds d'un cluster par exemple. Si k est le nombre de noeuds, alors nous pouvons diviser approximativement par k le temps d'exécution nécessaire pour ces étapes, ce qui permettrait d'obtenir une accélération non négligeable. Supposons que nous ayons 36 noeuds dans un cluster et que chaque noeud ait une mémoire interne de 1 Go. La première segmentation partielle durerait approximativement 40 secondes et la seconde étape plus ou moins 30 secondes. Cependant, le temps d'exécution de la segmentation finale resterait inchangé car celle-ci doit être faite séquentiellement dans un noeud. Globalement, le nouveau temps d'exécution serait de 21 minutes au lieu d'une heure. En tenant compte du temps de communication entre chaque noeud, nous pouvons espérer une accélération de 3 sur un tel cluster.

Chapitre 6

Validation de l'algorithme LSGRM et expérimentations

6.1 Étude de la stabilité du LSGRM

Nous rappelons qu'un algorithme échelonnable est stable s'il garantit des résultats identiques à ceux obtenus avec l'algorithme de référence. Dans notre situation, l'algorithme échelonnable représente l'algorithme LSGRM, noté A_{sca}, et l'algorithme de référence l'algorithme GRM, noté A_{ref}.
Dans le chapitre 1 section 1.6, nous avons expliqué comment démontrer que A_{sca} est stable. Nous rappelons les notations utilisées : E représente les pixels de l'image initiale, S_{ref} l'ensemble des segments résultants obtenus avec A_{ref} et S_{sca} l'ensemble des segments résultants obtenus avec A_{sca}. Avec ces notations, A_{sca} est stable si $\forall E$, nous avons :

$$\begin{cases} |S_{ref}| = |S_{sca}| \\ \forall s_j \in S_{sca}, \exists!\ s_i \in S_{ref} \text{ tel que } I(s_i) = s_j \text{ avec } s_j = s_i \end{cases} \tag{6.1}$$

où $I : S_{ref} \rightarrow S_{sca}$ est la fonction identité et E un ensemble de données en entrée.

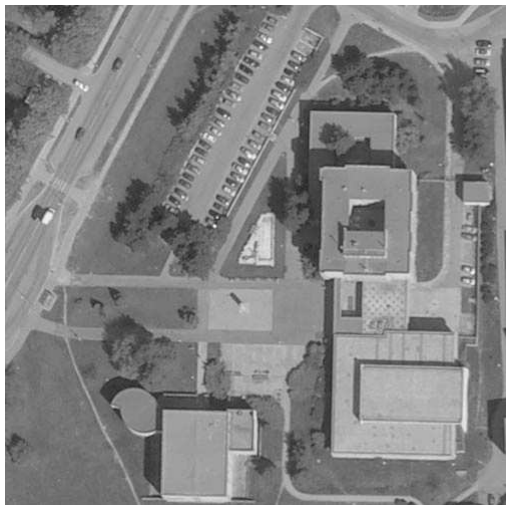
Dans notre contexte, l'ensemble des données en entrée E est composé de l'image à segmenter, et des paramètres specifiques au critère d'homogénéité utilisé pour la segmentation. Pour valider la stabilité, nous avons choisi d'utiliser les métriques de Hoover [62] en montrant que $RC = 1$, $RA = 0$, $RF = 0$ et $RM = 0$ à l'issue de la comparaison de la segmentation obtenue avec A_{sca} et celle obtenue avec A_{ref}. Pour cela, nous avons considéré un seuil $t = 1$ pour être intransigeant sur l'égalité des segments.

Pour ce faire, nous avons considéré deux images que nous notons I_1 et I_2 de taille 500×500 pixels. Ces images sont représentées sur la figure 6.1.

Trois critères d'homogénéité sont utilisés avec l'algorithme GRM :

- Critère d'homogénéité basé sur la distance euclidienne spectrale [48], noté DE.
- Critère Full Lambda Schedule [52], noté FLS.

(a) I_1



(b) I_2

Figure 6.1 – Images issues des satellites Ikonos et Quickbird présentant des zones ubaines.

- Critère de Baatz & Schäpe [45], noté BS.

La première expérience consiste à montrer que pour des marges de valeurs inférieures à la marge de stabilité, les segments résultants sont différents de ceux obtenus avec la segmentation de l'image complète. La procédure, illustrée par la figure 6.2, est composée de deux étapes :

1. L'image I_k avec $k = \{1, 2\}$ est segmentée en une seule fois pour fournir $S_{ref}(I_k)$. Le nombre d'itérations nécessaires pour effectuer la segmentation est égal à 32 itérations. Une tuile notée GT de taille 250×250 pixels est ensuite extraite de $S_{ref}(I_k)$ pour constituer la segmentation de référence GT.

2. Une tuile couvrant la même région que GT dans l'image est extraite à partir de I_k. Pour cette tuile, des marges comprises dans l'intervalle [0 : 125] avec un pas de 25 sont considérées. Nous notons la liste des tuiles segmentées résultantes $TS_0, TS_{25}, \ldots, TS_{125}$. Chacune de ces tuiles avec sa marge est segmentée complètement. Nous extrayons la région correspondant à GT et nous la comparons à GT en utilisant la métrique de Hoover.

Seul le score de l'instance RC nous importe puisqu'elle représente la proportion de segments identiques entre les deux segmentations. Pour les deux images, la figure 6.3 montre l'évolution du score de RC en fonction de la valeur de la marge pour un critère d'homogénéité donné.

En analysant les graphes dans la figure 6.3, nous identifions un comportement similaire pour chaque critère et pour chaque image. En effet, le score de RC croît avec

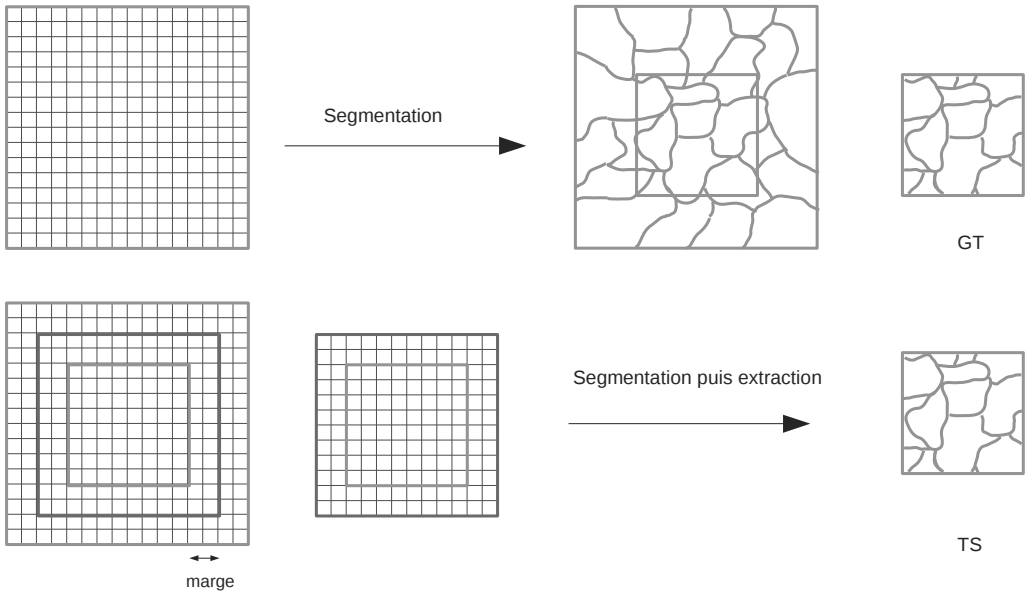


Figure 6.2 – Protocole pour étudier l'évolution de RC en fonction de la valeur de la marge.

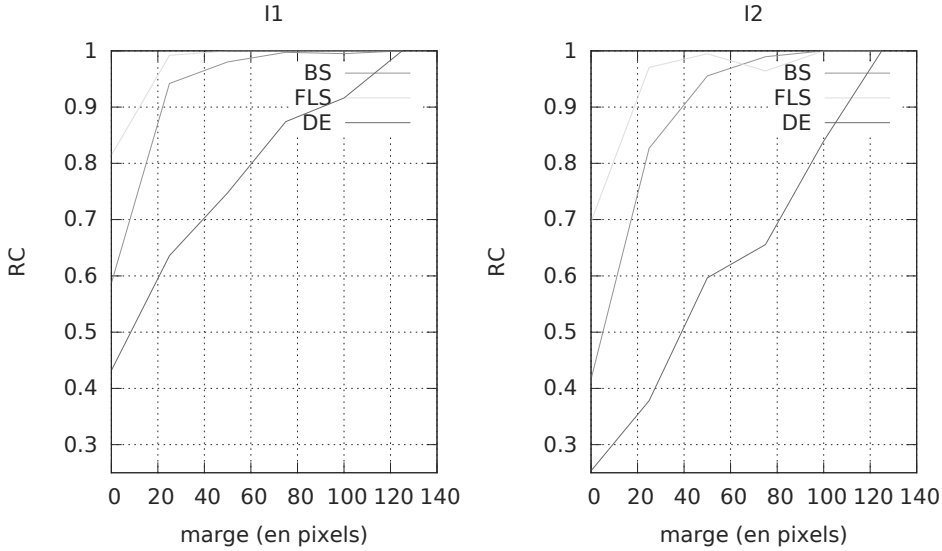


Figure 6.3 – Évolution de l'instance RC en fonction de la marge exprimée en pixels

l'augmentation de la valeur de la marge pour converger vers 1. En revanche, la vitesse de convergence dépend du critère d'homogénéité mais aussi de la nature de l'image. Pour l'image I_1 et le critère FLS, une marge de 50 pixels est suffisante pour obtenir un score

$RC = 1$. De même pour l'image I_2 et le critère BS, une marge de 100 pixels est suffisante pour obtenir un score $RC = 1$. Par contre, pour le critère DE, il est nécessaire de considérer une marge de 125 pixels c'est-à-dire toute l'image pour obtenir un score $RC = 1$. Lorsque nous comparons les évolutions de RC pour chaque image, nous remarquons que l'ordre relatif de convergence est similaire entre les critères. Cependant, pour un critère donné et différentes images, le score RC n'est pas le même pour des marges identiques. Par exemple, pour le critère FLS, une marge de 50 pixels est suffisante pour I_1 mais ce n'est pas le cas pour I_2.

D'après cette analyse, nous pouvons conclure qu'il s'avère difficile d'anticiper la valeur exacte de la marge suffisante car elle dépend à la fois du critère d'homogénéité et de l'image. C'est la raison pour laquelle nous avons déterminé une expression générique de la marge de stabilité qui assure pour n'importe quel critère et n'importe quelle image un résultat identique à celui obtenu sans tuilage.

Pour valider l'expression de la marge, la procédure suivante a été effectuée :

1. L'image I_k avec $k = \{1, 2\}$ est segmentée en une seule fois pour fournir $S_{ref}(I_k)$ qui représente GT.

2. L'image I_k est d'abord divisée en 4 tuiles de taille 250×250 pixels. Pour chaque tuile une marge de stabilité M_n est considérée avec $n \in [1, ..., 6]$ pour des marges allant de 2 à 126 pixels. Chaque tuile est ensuite segmentée séparément en appliquant n itérations. Les segments instables sont ensuite supprimés de chaque graphe (voir le chapitre 5 section 5.2) et les graphes sont regroupés pour former le graphe global de l'image I_k (voir le chapitre 5 section 5.5). La segmentation est alors achevée sur ce graphe pour former TS. Les scores RC sont ensuite obtenus en comparant GT et TS avec la métrique de Hoover.

Sans surprise, nous avons obtenu un score $RC = 1$ pour chaque valeur correcte de la marge de stabilité et cela pour les 2 images et les 3 critères. Nous avons ainsi réussi à définir une marge de stabilité universelle qui est correcte pour n'importe quel critère, n'importe quelle heuristique de fusion et n'importe quelle image. Dans la suite, nous étudions le comportement de l'algorithme échelonnable lorsque la mémoire interne disponible est limitée et nous montrons l'évolution des graphes intermédiaires des tuiles au cours de la procédure.

6.2 Segmentation d'une scène complète Pléiades

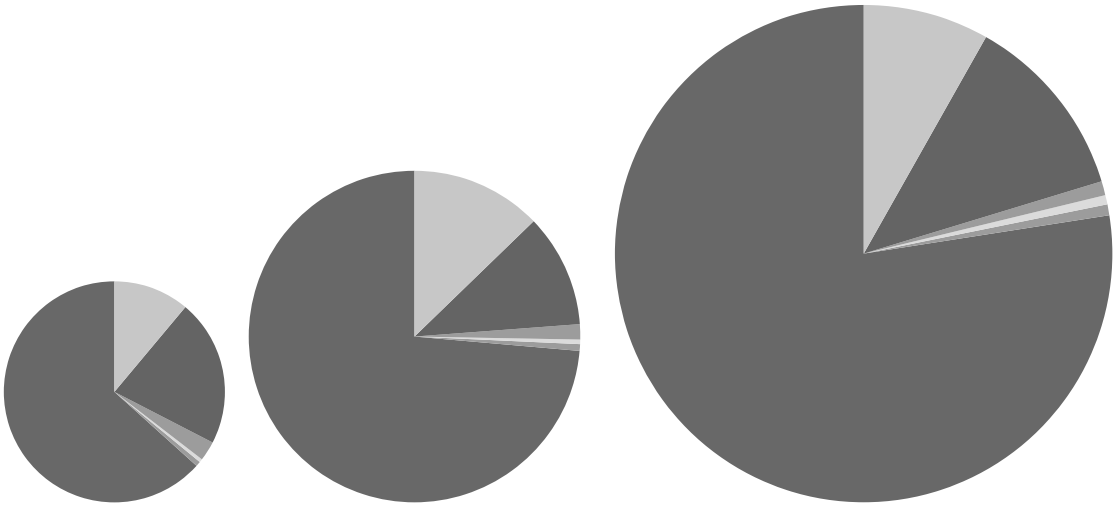
Dans cette section, nous appliquons notre algorithme LSGRM pour segmenter des images Pléiades THR de très grande taille. Pour cela, nous avons utilisé une scène Pléiades de taille 14000×14000 pixels de la ville Commerce City dans la Colorado aux États-Unis. Cette image multispectrale possède 4 bandes spectrales et a une résolution de 50 cm. Nous avons utilisé le critère local d'homogénéité de Baatz & Schäpe avec les paramètres $w_c = 0,7$, $w_s = 0.3$ et $scale = 40$. Nous avons effectué 3 segmentations sur des extraits de taille 6000×6000, 9000×9000 et 14000×14000 pixels. Pour

l'ensemble des segmentations, nous avons simulé un ordinateur avec une capacité de 6 Go en mémoire interne. Pour l'extrait de taille 6000×6000 pixels, nous avons découpé l'image en tuiles de taille 2000×2000 avec une marge de stabilité égale à 510 pixels, ce qui correspond à 8 itérations. Pour le second extrait, nous avons découpé l'image en tuiles de taille 2250×2250 pixels avec une marge de stabilité égale à 510 pixels. Enfin pour la segmentation de l'image complète, nous avons utilisé des tuiles de taille 2800×2800 pixels avec une marge égale à 1022 pixels correspondant à 9 itérations. Pour chaque segmentation, une seule segmentation partielle était suffisante pour stocker le graphe des segments en mémoire interne. Le tableau 6.1 représente l'évolution du temps d'exécution en fonction de la taille de l'image. La figure 6.4 récapitule la répartition du temps d'exécution pour chaque étape de l'algorithme LSGRM en fonction de la taille de l'image.

Taille de l'image	Mémoire	Taille des tuiles	Marge	Temps d'exécution
6000×6000	576 Mo	2000×2000	510	0h50
9000×9000	1.3Go	2250×2250	510	2h05
14000×14000	3.1Go	2800×2800	1022	7h00

Table 6.1 – Évolution du temps d'exécution en fonction de la taille de l'image en entrée.

Puisqu'il n'est pas possible d'afficher toute l'image segmentée, la figure 6.5 montre l'extrait d'une zone segmentée située sur une bordure commune entre 2 tuiles. Nous observons l'absence d'artefacts sur la bordure grâce aux marges de stabilité utilisées lors des segmentations partielles. Cette expérience montre la capacité de la solution proposée à segmenter des images à très haute résolution tout en garantissant des résultats identiques à ceux que nous aurions obtenu sans tuilage.



(a) 6000×6000 **(b)** 9000×9000 **(c)** 14000×14000

Segmentations partielles

Écriture des graphes

Suppression des segments dupliqués et mise à jour des arêtes

Suppression des segments instables

Regroupement des graphes

Segmentation finale

(d)

Figure 6.4 – Répartition du temps d'exécution pour chaque étape de l'algorithme LSGRM.

(a) Image originale de taille 14000×14000 pixels

(b) Vue sur la bordure entre 2 tuiles

(c) Vue des segments sur la bordure

Figure 6.5 – Vue sur une zone contenant la bordure entre 2 tuiles, délimitée par un rectangle rouge dans la vue de l'image complète (figure 6.5a). La figure 6.5b représente la zone dans l'image complète et la figure 6.5c la même zone segmentée. La bordure de la tuile passe au mileu de cette zone.

Chapitre 7

Bilan sur la segmentation échelonnable

7.1 Conclusions

Cette première partie de la thèse s'est attachée à étudier la problématique de la segmentation de larges images en télédétection ne pouvant être stockées dans la mémoire interne d'un ordinateur. En optant pour une stratégie de tuilage, il a été observé la présence d'artefacts sur les bordures communes des tuiles adjacentes. Ces artefacts traduisent la discontinuité des surfaces de contact des segments situés de part et d'autre de ces bordures. De plus, une comparaison entre les segmentations avec tuilage et sans tuilage a mis en évidence la présence de segments différents situés à l'intérieur des tuiles. Nous avons alors conclu que le tuilage influence de manière globale la qualité des segments résultants. Nous avons étudié une solution permettant d'utiliser le tuilage tout en garantissant des résultats sans la présence d'artefacts sur les bordures des tuiles.

Le chapitre 2 s'est attaché à présenter les différentes méthodes de segmentation par fusion de régions. Différents critères d'homogénéité ont été décrits. Certains sont basés sur l'information spectrale, d'autres sur l'information spatiale ou sur la combinaison des deux types d'information. Nous avons également présenté différentes heuristiques de fusion de régions et nous avons mis en évidence les qualités de l'heuristique LMBF ("Local Mutual Best Fitting") qui permet de s'affranchir de l'ordre de visite des segments. Nous avons identifié des étapes génériques pour la segmentation par fusion de régions, ce qui nous a motivé à développer un algorithme de fusion de regions générique, nommé algorithme GRM ("Generic Region Merging"). Ce nouvel algorithme est indépendant du critère local d'homogénéité et de l'heuristique de fusion.

Le chapitre 3 décrit l'algorithme GRM qui constitue notre algorithme de référence pour l'étude du passage à l'échelle des méthodes par fusion de régions. Dans cet algorithme, les segments de l'image sont représentés sous la forme d'un graphe d'adjacence où chaque segment est un noeud du graphe et chaque lien d'adjacence entre deux segments une arête. Stocker les coordonnées des pixels dans chaque segment est problématique

pour le passage à l'échelle de l'algorithme. C'est pourquoi, nous avons proposé une nouvelle représentation pour localiser les segments en se basant sur le déplacement le long de leur contour. N'existant que 4 déplacements possibles, chacun peut être représenté en n'utilisant que 2 bits. Cela a donc permis de réduire significativement la taille du graphe de segments en mémoire interne.

Une fois l'algorithme de référence établi, le chapitre 4 s'est attaché au processus de stabilisation de celui-ci. Une zone d'influence de chaque segment S a été déterminée et représente l'ensemble des segments qui peuvent potentiellement interagir avec S lors d'une itération. À partir de cette zone d'influence, il a été possible de déterminer une marge de stabilité autour de chaque tuile. La valeur de la marge de stabilité dépend du nombre d'itérations que nous voulons appliquer et représente la largeur de la couronne de stabilité autour de la tuile.

Le chapitre 5 propose un algorithme échelonnable basé sur l'algorithme GRM. Cet algorithme échelonnable consiste à appliquer une succession de segmentations partielles des tuiles jusqu'à ce que l'une des deux conditions suivantes soit vérifiée :

- l'ensemble des segments de l'image peut être stocké en mémoire interne.
- La segmentation est terminée dans chaque tuile (plus aucune fusion n'est possible dans chaque tuile).

La deuxième condition est intéressante car elle signifie que l'algorithme échelonnable peut être appliqué sur des images de dimension arbitraire car il n'est pas nécessaire que le graphe des segments résultants de l'image puisse être stocké en mémoire interne.

Finalement, le chapitre 6 a permis de valider l'expression de la marge de stabilité en étudiant la stabilité de l'algorithme échelonnable avec la métrique de Hoover. En particulier, il a été confirmé que l'algorithme échelonnable proposé garantit des segments résultants identiques à ceux obtenus avec l'algorithme GRM sans tuilage. La faisabilité de l'algorithme échelonnable a été vérifiée par la segmentation de plusieurs scènes Pléiades avec un ordinateur dont la capacité de la mémoire interne est limitée. En particulier, il a été observé que l'étape de segmentation prédomine par rapport aux accès en lecture et écriture dans la mémoire externe. Cette opération étant parallélisable, cela signifie qu'un gain important peut être obtenu en adaptant cet algorithme dans un environnement distribué et/ou parallèle.

7.2 Perspectives

Dans ces travaux, l'objectif était de trouver une solution algorithmique pour pallier la contrainte de la mémoire et permettre la segmentation d'images satellite de taille arbitraire. L'algorithme LSGRM proposé remplit cet objectif.

Après avoir analysé les temps d'exécution des étapes de l'algorithme LSGRM, nous avons conclu que les temps liés aux accès en lecture et écriture dans la mémoire externe sont négligeables par rapport aux temps d'exécution liés à la segmentation des graphes et à l'ajout des marges de stabilité lors des segmentations partielles. Puisque ces 2 étapes

sont facilement parallélisables et qu'elles prédominent au niveau du temps d'exécution, une idée d'amélioration serait de paralléliser ces tâches. Dans un premier temps, une parallélisation sur un seul CPU pourrait être étudiée en partageant la quantité de mémoire interne à chaque processus. Une seconde idée serait d'utiliser les GPUs en adaptant le code et en portant une attention particulière sur la minimisation du transfert des données entre le CPU et le GPU. Enfin, une troisième étape serait d'adapter l'algorithme LSGRM dans un environnement distribué et parallèle. Dans cette dernière perspective, une attention particulière doit être portée sur les temps de communication entre chaque noeud du cluster.

Nous avons choisi certaines structures de données pour l'algorithme GRM et sa version échelonnable LSGRM. Une étude plus appronfondie pourrait être faite sur le choix des structures de données à utiliser en établissant une étude comparative selon les architectures des mémoires cache.

Enfin, nous avons généralisé notre solution à la famille des algorithmes par fusion de régions. L'approche dans [63] peut-être intégrée dans notre solution car de nombreuses étapes sont communes. Une idée serait d'étendre cette approche à d'autres familles d'algorithmes de segmentation comme les méthodes Superpixels, Split & Merge, *etc.*

Classification

Chapitre 8

Contexte des travaux

8.1 Classification en télédétection

D'après la définition générale de Cournot (1851), la classification est un processus qui permet de grouper des objets ayant des caractéristiques communes et de donner aux groupes génériques ainsi formés une étiquette ou un nom générique.
En télédétection, elle consiste à assigner à chaque pixel ou chaque ensemble de pixels dans une image la classe de couverture ou d'usage à laquelle il appartient. Des exemples de classe peuvent être une zone agricole, la forêt, une zone urbaine, *etc.* L'image résultante d'une classification représente ainsi une carte d'occupation des sols.
La classification en télédétection a émergé dans les années 1980 et la plupart des méthodes proposées dans ces années effectuaient une analyse basée sur les pixels des images où chaque pixel se voit assigner une étiquette identifiant la classe à laquelle il appartient. Cependant, la classification basée sur les pixels a connu certaines limitations avec l'apparition des images satellite à très haute résolution dans la mesure où un objet dans l'image peut contenir des pixels spectralement hétérogènes. Ainsi, au lieu de considérer l'analyse de chaque pixel indépendamment des autres, des méthodes récentes de classification proposent d'analyser des objets constitués d'un ensemble de pixels connectés ayant des propriétés spectrales et spatiales homogènes [26, 77, 78]. Une revue complète des différentes techniques de classification en télédétection est faite dans [79].

Dans nos travaux, nous considérons des données caractérisées par des attributs numériques qui peuvent correspondre à des valeurs radiométriques, statistiques ou géométriques. Chacune des données représente ainsi un vecteur appartenant à un espace $\mathbb{R}^F$ où F est le nombre d'attributs.
Sur la base des attributs de chaque donnée, l'objectif de la classification est de déterminer la classe d'une donnée parmi l'une des K classes définie par un ensemble fini d'étiquettes $C = \{C_1, \ldots, C_K\}$ où C_k représente un entier positif.
Ainsi, à partir d'une image (multi-spectrale, hyperspectrale, multi-temporelle, *etc*) acquise, nous obtenons une image étiquetée où chaque pixel contient une valeur dans C qui représente sa classe.

Deux types d'approche de classification sont distingués dans la littérature : les méthodes de classification non supervisée sans aucune connaissance *a priori* sur les données et les méthodes de classification supervisée.

8.1.1 Classification non supervisée

En classification non supervisée, l'appartenance des données à l'une des classes dans C n'est pas connue. La classification non supervisée consiste à partitionner automatiquement les données par similarité. Les classes sont ainsi reconnues *a posteriori*. Il existe de nombreuses méthodes de classification non supervisée dont les principales sont les méthodes ascendantes hiérarchiques [80], la classification K-Means [81] et le maximum de vraisemblance [82]. Les deux premières méthodes sont itératives et consistent à déterminer dans $\mathbb{R}^F$ le centre de gravité des grappes de données représentant les différentes classes. Pour mesurer la similarité entre les données, ces méthodes utilisent des métriques au sens d'une distance entre les vecteurs d'attributs. La troisième méthode se place dans un cadre probabiliste. Elle consiste à définir des populations homogènes en se basant sur les distributions de probabilité. La méthode classique pour estimer les densités de probabilité dans un ensemble de données est la méthode d'espérance-maximisation (EM). Elle considère que les données appartiennent à K classes qui sont représentées par un mélange de distributions (souvent gaussiennes) de paramètre $\theta = [\theta_1, \ldots, \theta_K]$. En considérant un paramètre initial θ^0, la méthode EM est une procédure itérative qui converge vers θ^f en maximisant à chaque itération la vraisemblance.

8.1.2 Classification supervisée

La classification supervisée consiste à déterminer une fonction de décision à partir d'un ensemble de données d'apprentissage. Un ensemble d'apprentissage contient des données pour lesquelles la classe est connue. Les ensembles d'apprentissage peuvent être constitués par des observations faites sur le terrain (vérités terrain), par des connaissances thématiques obtenues auprès d'un expert ou par des bases de données de référence comme, par exemple, celles produites par l'IGN (Institut national de l'information géographique et forestière). Une phase d'apprentissage est ainsi réalisée pour construire une fonction de décision capable de distinguer les classes à partir des attributs. Par exemple dans le laboratoire du CESBIO, de nombreux ensembles d'apprentissage sont disponibles. En particulier sur la zone du Sud Ouest de la France, le CESBIO dispose de longues séries multi temporelles avec 48 dates d'images Landsat 8 et 24 dates d'images SPOT 4 (Take 5) afin de simuler les images Sentinel-2. Ces données sont décrites en dans l'annexe B. À ces images est associé un registre parcellaire géographique qui permet d'obtenir la classe d'un échantillon de pixels dans la série. La classification supervisée permet d'obtenir des modèles de prédiction plus performants, comparés à ceux obtenus avec la classification non supervisée. C'est pourquoi, nous avons choisi d'étudier le passage à l'échelle des méthodes d'apprentissage supervisé. Les nouveaux algorithmes seront testés en utilisant cette base de données d'apprentissage disponible au CESBIO.

8.2 Revue des méthodes de classification supervisée

Les méthodes de classification supervisée sont utilisées en télédétection pour la production de cartes d'occupation des sols et la reconnaissance d'objets. Différentes approches existent pour établir une fonction de décision à partir d'un ensemble de données étiquetées. Par la suite, nous considérons que E constitue un ensemble d'apprentissage dans $\mathbb{R}^F$ de taille N. Chaque donnée est représentée dans $\mathbb{R}^F$ sous la forme d'un vecteur et est notée x_i avec $i \in [1, \ldots, N]$. À chaque donnée est associée une étiquette notée $y_i \in C$ qui identifie la classe de x_i.

8.2.1 Classification bayésienne

Cette approche [83] utilise un modèle probabiliste où les données sont associées à une variable aléatoire. Nous notons X_i la variable aléatoire associée à la donnée x_i. x_i représente une réalisation possible de X_i. La classification consiste à chercher pour chaque donnée la réalisation y_i d'une variable aléatoire représentant la classe. Pour cela, nous cherchons à maximiser la probabilité conditionnelle notée $P(Y_i = y_i|X_i = x_i)$ en utilisant le théorème de Bayes :

$$P(Y_i = y_i|X_i = x_i) = \frac{P(X_i = x_i|Y_i = y_i) \times P(Y_i = y_i)}{P(X_i = x_i)} \tag{8.1}$$

où $P(Y_i = y_i)$ est la probabilité d'apparition des classes C_i, $P(X_i = x_i|Y_i = y_i)$ est le terme de vraisemblance et $P(X_i = x_i)$ est la probabilité d'apparition des primitives x_i. En faisant abstraction de $P(Y_i = y_i)$ et $P(X_i = x_i)$, conclure que $x_i \in y_i$ revient à déterminer :

$$x_i \in y_i \Leftrightarrow P(X_i = x_i|Y_i = y_i) > P(X_i = x_i|Y_j = y_j), \forall j \neq i \tag{8.2}$$

Ainsi, l'étape critique dans la classification bayésienne est de déterminer le terme de vraisemblance. En imagerie optique, il est généralement assumé que celui-ci suit une densité de probabilité gaussienne [84] de paramètres (μ, σ) qui sont déterminés à partir de l'ensemble des données d'apprentissage avec la méthode EM.

8.2.2 Les réseaux de neurones

Cette approche [85] est inspirée par le fonctionnement d'un système nerveux biologique. Un réseau de neurones est constitué d'un ensemble de couches connectées. Ainsi, un nœud est connecté avec un ou plusieurs nœuds dans la couche précédente et avec un ou plusieurs nœuds dans la couche suivante. Ce nœud peut aussi être connecté avec un ou plusieurs nœuds dans la même couche. Pour résumer, un réseau de neurones est composé d'une couche en entrée, d'une ou plusieurs couches cachées et d'une couche en sortie. La

structure générale d'un réseau de neurones est illustrée par la figure 8.1. Chaque nœud dans la couche cachée calcule la combinaison linéaire des valeurs reçues en entrée de la forme $o = \sum_{i=1}^{n} w_i s_i$ où w_i est le poids attribué au nœud i dans la couche précédente, puis applique une fonction d'activation non-linéaire (comme par exemple une fonction sigmoïde) pour produire sa sortie. La phase d'apprentissage consiste à déterminer les poids à attribuer pour chaque neurone dans les couches cachées. Pour le perceptron et la plupart des réseaux de neurones, il y a autant de neurones dans la couche de sortie que de classes. Chaque neurone va donner une probabilité d'appartenance à chaque classe et nous considérons la probabilité d'appartenance maximum lors de la prédiction d'une nouvelle observation.

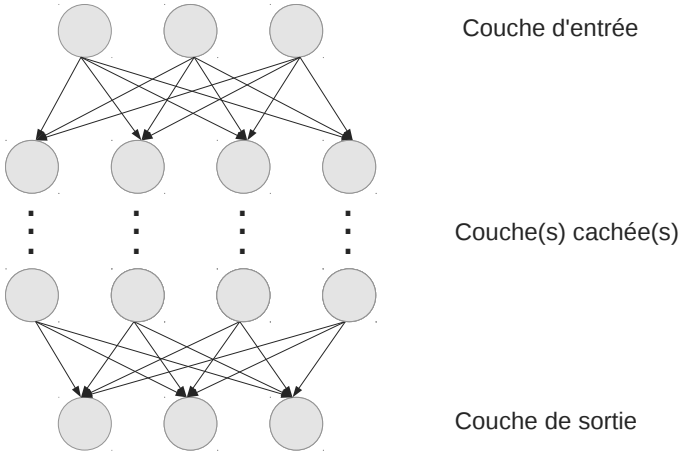


Figure 8.1 – Topologie d'un réseau de neurones de type perceptron multi-couche.

8.2.3 Les séparateurs à vaste marge

Les séparateurs à vaste marge sont des classifieurs binaires [86] qui déterminent l'équation d'un hyperplan optimal qui sépare les données suivant leur classe. Supposons un ensemble de données d'apprentissage $E = \{x_i\}_{i=1}^{N}$ avec $x_i \in \mathbb{R}^F$ et leurs étiquettes $y_i \in \{-1, +1\}$. L'équation de l'hyperplan qui sépare les données de manière optimale est de la forme $f(\vec{x}) = \vec{w} \cdot \vec{x} + b = 0$. Cet hyperplan séparateur, noté $\mathbb{H}^*$, est celui qui maximise la marge de séparation entre les données de classe positive et les données de classe négative. Sa représentation est illustrée par la figure 8.2 dans le cas où $F = 2$.

Ainsi, l'équation de $\mathbb{H}^*$ doit vérifier les contraintes suivantes pour une donnée x_i avec une classe y_i :

$$y_i(\vec{w} \cdot \vec{x_i} + b) \geq 1, \forall i \in [1, N]; \tag{8.3}$$

ce qui signifie que les données avec les classes +1 et -1 sont situées de part et d'autre de l'hyperplan. La prédiction d'une nouvelle donnée x s'effectue en étudiant le signe de la fonction de décision $f(x) = \vec{w} \cdot \vec{x} + b$. Si $f(x)$ est positif alors la donnée appartient à la classe positive sinon la donnée appartient à la classe négative.

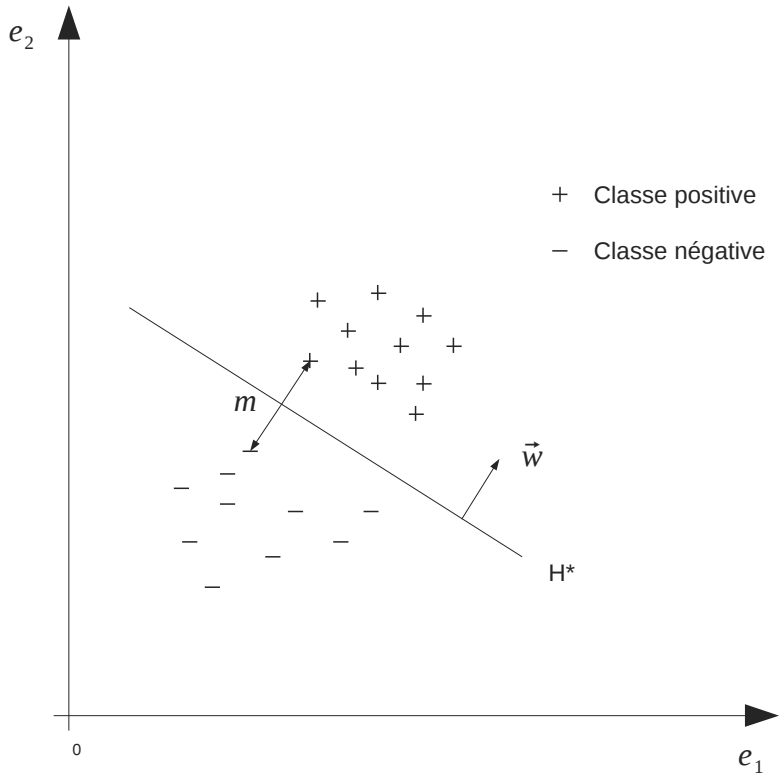


Figure 8.2 – Hyperplan séparant les données d'apprentissage de manière optimale au sens de la maximisation de la marge dans $\mathbb{R}^2$.

Considérer seulement cette contrainte ne suffit pas car il existe une infinité d'hyperplans qui sont solution. Intuitivement, un hyperplan situé trop près des données d'apprentissage n'aura pas une bonne capacité de généralisation pour la prédiction de nouvelles données. Ainsi, une seconde contrainte est ajoutée et consiste à maximiser la distance des données les plus proches de l'hyperplan. Cette distance est nommée la marge et est notée m dans la figure 8.2. Les données d'apprentissage les plus proches de l'hyperplan sont les vecteurs supports. Nous fixons le vecteur normal $\vec{w}$ et la valeur de biais b de telle façon que $\vec{w} \cdot \vec{x} + b = \pm 1$.
Le problème de maximisation de la marge peut-être déterminé géométriquement. En notant x_- et x_+ deux vecteurs support de chaque classe, la marge peut-être exprimée comme le vecteur différence $(\vec{x_+} - \vec{x_-})$ projeté sur le vecteur normal unitaire $\frac{\vec{w}}{||\vec{w}||}$:

$$m = (\vec{x_+} - \vec{x_-}) \cdot \frac{\vec{w}}{||\vec{w}||}; \tag{8.4}$$

en notant que $\vec{w} \cdot \vec{x_+} = 1 - b$ et $\vec{w} \cdot \vec{x_+} = -1 - b$, nous avons par simplification :

$$m = \frac{2}{||\vec{w}||}. \tag{8.5}$$

Par conséquent, maximiser la marge revient à minimiser la norme du vecteur normal à l'hyperplan. Sans perdre de généralité, le problème de maximisation est transformé en un problème quadratique convexe qui a la propriété d'obtenir un minimimum global. La fonction à minimiser a alors pour expression $\frac{1}{2}||\vec{w}||^2$.
Il existe plusieurs méthodes pour résoudre ce problème d'optimisation, la plus connue étant celle utilisant les multiplicateurs de Lagrange en associant à chaque donnée d'apprentissage un multiplicateur. Plusieurs méthodes ont été proposées dans la littérature [87, 88] pour déterminer l'équation de $\mathbb{H}^*$.

A noter que lorsque les données d'apprentissage ne sont pas linéairement séparables, les contraintes peuvent être modifiées en utilisant une variable d'erreur notée $\xi_i \geq 0$:

$$\vec{w} \cdot \vec{x_i} + b \geq 1 - \xi_i \text{ si } y_i = 1; \tag{8.6}$$

$$\vec{w} \cdot \vec{x_i} + b \leq -1 + \xi_i \text{ si } y_i = -1. \tag{8.7}$$

Si $\xi_i > 1$ alors la donnée d'apprentissage est mal classée. Dans la fonction à minimiser, nous introduisons un second terme pour minimiser les erreurs ξ_i de la façon suivante :

$$\min \left(\frac{1}{2}||\vec{w}||^2 + C \sum_i \xi_i \right); \tag{8.8}$$

où C représente un paramètre de tolérance pour les erreurs.

Lorsque la surface de décision est non linéaire, l'introduction des variables d'erreur ne s'applique pas. La solution est de retranscrire les données d'apprentissage dans un espace de plus grande dimension (éventuellement infinie) pour trouver des séparateurs linéaires. Des fonctions noyaux [89] sont alors utilisées pour représenter une mesure de similarité dans de tels espaces.

Enfin, l'apprentissage peut être étendu à un ensemble de données contenant plus de deux classes en adoptant deux stratégies [90]. La première est le "un contre tous" qui résulte à K classifieurs binaires s'il y a K classes. Chaque classifieur identifie l'apprentissage à une classe par rapport à toutes les autres. Une autre stratégie est d'effectuer du "un contre un" ce qui résulte à $K \times (K-1)$ classifieurs binaires qui identifient l'appartenance à l'une des 2 classes pour chaque paire de classes possible. Un vote majoritaire est ensuite effectué.

8.2.4 Les arbres de décision

Cette approche permet d'obtenir un arbre qui combine logiquement une séquence de tests. Chaque test constitue une comparaison de la valeur d'un attribut numérique d'une donnée d'apprentissage avec un seuil. Géométriquement, construire un arbre de décision revient à partitionner l'espace des attributs des données en régions où chaque région représente une classe. Lors de la prédiction, lorsqu'une donnée se situe dans cette région alors l'arbre de décision lui assigne la classe correspondante.

Il existe différentes méthodes dans la littérature pour construire un ou plusieurs arbres de décision à partir d'un ensemble de données d'apprentissage. L'objectif commun

de chaque méthode est de déterminer la séquence de tests optimale pour partitionner l'espace des attributs en régions homogènes.

Dans [91, 92], les auteurs proposent les algorithmes CART ("Classification And Regression Trees") et C4.5. Ces deux méthodes construisent un arbre de décision qui est composé soit de nœuds contenant un test, nommés nœuds décisionnels, soit de nœuds indiquant la classe de la région dans l'espace des attributs, nommés noeuds feuille car ils sont situés au niveau des feuilles de l'arbre. La structure d'un arbre de décision est représentée dans la figure 8.3.

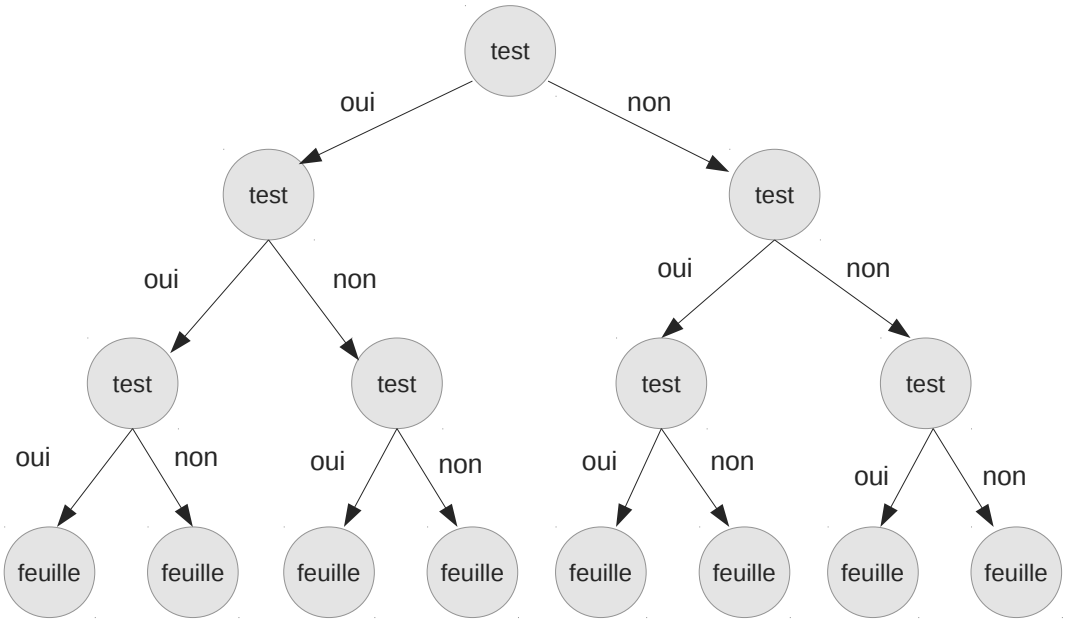


Figure 8.3 – Structure d'un arbre de décision valable pour CART et C4.5

Le test dans chaque nœud décisionnel est effectué de la façon suivante. Les données en entrée sont triées suivant l'attribut A. La séquence des valeurs triées est notée $(v_1, v_2, \ldots, v_m)$. Toutes les coupures possibles sont ensuite analysées, il y en a $m-1$ pour m valeurs. Une coupure possible divise la séquence de départ en deux nouvelles séquences $S_l = (v_1, \ldots, v_i)$ et $S_r = (v_{i+1}, \ldots, v_m)$.
Dans [91], les auteurs définissent une mesure d'impureté basée sur l'entropie d'une population P de données :

$$I(P) = -\sum_{k=1}^{K} \frac{m_k}{m} \times \log_2 \frac{m_k}{m}; \tag{8.9}$$

où m_k est le nombre de données appartenant à la classe C_k dans la population.
La coupure sélectionnée pour un nœud décisionnel est celle maximisant le gain d'information, qui est la différence d'entropie entre la population initiale S et les deux nouvelles populations S_l et S_r :

$$gain(S, S_l, S_r) = I(S) - (I(S_l) + I(S_r)) \,. \tag{8.10}$$

Le seuil pour le test du nœud décisionnel résultant est alors $\frac{v_i+v_{i+1}}{2}$. Le nœud devient une feuille lorsque les données en entrée appartiennent à la même classe.
Dans [92], les auteurs proposent une mesure d'impureté basée sur le critère de Gini :

$$Gini(P) = 1 - \sum_{k=1}^{K} \frac{m_k^2}{m^2}. \tag{8.11}$$

La coupure sélectionnée est celle maximisant la différence d'impureté entre S et les populations S_r et S_l :

$$\Delta I(S, S_l, S_r) = Gini(S) - Gini(S_l) - Gini(S_r). \tag{8.12}$$

À la différence de CART, C4.5 permet d'utiliser des attributs catégoriques pour établir des tests basés sur l'appartenance à des collections. Les règles de décision sur des attributs catégoriques peuvent engendrer plusieurs coupures et ainsi la création de plusieurs nœuds fils. De plus, C4.5 permet de traiter les données manquantes en se basant sur la probabilité d'apparition des données connues. Enfin, C4.5 propose une stratégie d'élagage de l'arbre pour le rendre moins complexe et résoudre le problème du sur-entraînement tout en essayant de conserver ses performances prédictives. Cette opération consiste à fusionner des nœuds décisionnels en minimisant le risque d'erreur et la complexité de l'arbre.

Dans [93], les auteurs proposent un classifieur plus efficace que les arbres de décision nommé forêt aléatoire. Une forêt aléatoire est un comité d'arbres de décision dans lequel a été introduit de l'aléa. L'aléa est introduit sur 2 niveaux. Chaque arbre est construit à partir d'un sous-ensemble de données d'apprentissage tiré aléatoirement à partir de l'ensemble des données initial. Ensuite, pour la construction de chaque nœud décisionnel, nous choisissons G attributs aléatoirement et nous sélectionnons, parmi ceux-ci, celui qui minimise le critère de Gini. La prédiction d'une nouvelle donnée s'effectue par un vote majoritaire des arbres de décision. Nous pouvons citer plusieurs autres classifieurs utilisant un comité d'arbres de décision comme le Tree Bagging [94], le random subspace [95], le Random Select Split [96] ou encore le Gradient Boosted Trees [97].

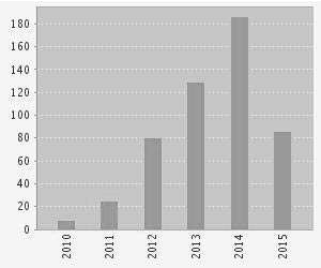
8.3 Problématique

Comme nous l'avons expliqué dans l'introduction de cette thèse, les récentes missions spatiales d'observation de la Terre produisent des grands volumes de données dont la taille dépasse la quantité de mémoire interne disponible dans les ordinateurs. Dans le cas de la classification supervisée, cette limitation impose une visibilité réduite des données d'apprentissage pouvant dégrader la performance de prédiction du classifieur. Comme pour la segmentation, un des défis actuels est de concevoir des méthodes d'apprentissage supervisé échelonnables capables de traiter des volumes de données de taille arbitraire.

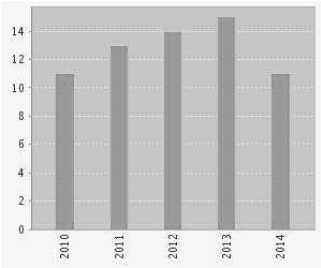
Dans la section précédente, nous avons établi une revue des principales méthodes de classification supervisée. Comme il était évidemment impossible de les considérer toutes

dans cette thèse, nous avons fait une étude bibliométrique pour analyser les fréquences d'utilisation de ces méthodes au cours des 5 dernières années dans le domaine de la télédétection. La figure 8.4 représente les fréquences de citations et de publications pour chaque méthode : bayes (classification bayésienne), nn (réseau de neurones), svm (séparateur à vaste marge), rf (forêt aléatoire) dans les conférences et journaux spécialisés dans le domaine de la télédétection. L'étude a été réalisée avec l'outil Web of Science. Nous précisons que l'année 2015 n'est pas terminée au moment de l'écriture de ce manuscrit (Août 2015) et nous montrons les fréquences de citations et de publications jusqu'au mois de Juin 2015. À partir des graphes, trois méthodes ressortent : les réseaux de neurones, les séparateurs à vaste marge et les forêts aléatoires. Particulièrement, nous notons une nette progression des fréquences de publications et de citations pour les forêts aléatoires. Ceci est dû à la performance des comités de classifieurs faibles, à la vitesse de calcul et la facilité de paramétrage.

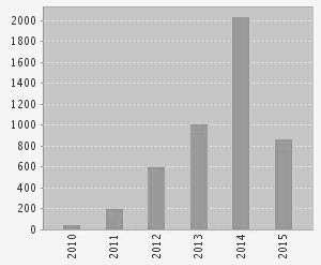
Nous proposons ainsi de nous intéresser par la suite à l'étude du passage à l'échelle de l'algorithme forêt aléatoire.



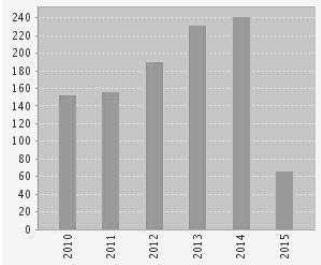
(a) bayes : citations



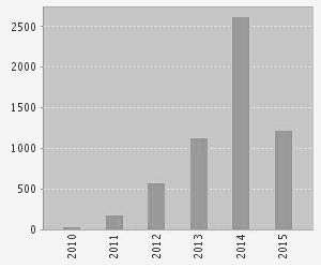
(b) bayes : publications



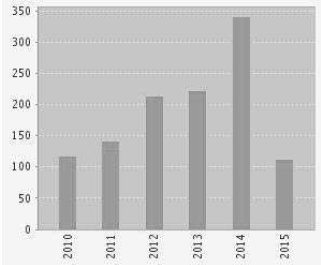
(c) nn : citations



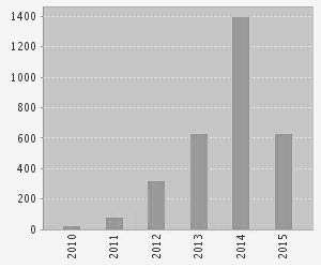
(d) nn : publications



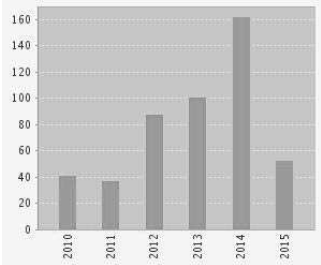
(e) svm : citations



(f) svm : publications



(g) rf : citations



(h) rf : publications

Figure 8.4 – Étude bibliométrique analysant la fréquence de citations et de publications pour chaque méthode. Puisque l'année 2015 n'est pas terminée au moment où ce manuscrit est écrit, nous montrons les fréquences de citations et de publications jusqu'au mois de Juin 2015.

Chapitre 9

Forêts aléatoires

9.1 Description générale de l'algorithme

Au chapitre précédent, une revuex des différentes méthodes d'apprentissage supervisé a été réalisée et nous avons retenu l'algorithme des forêts aléatoires dans le cadre de notre étude bibliométrique. L'algorithme, que nous décrivons par la suite, constitue l'algorithme de référence noté A_{ref} pour lequel il sera proposé un algorithme échelonnable.

Commençons par définir quelques notations utilisées dans la description de l'algorithme. Soit $E = \{x_i, y_i\}_{i=1}^{N}$ un ensemble de données d'apprentissage où une donnée est représentée par un vecteur d'attributs numériques $x_i \in \mathbb{R}^F$ et par sa classe $y_i \in C = \{C_1, \ldots, C_K\}$. T représente le nombre d'arbres de décision dans le comité. Théoriquement, plus T est important, meilleure est la prédiction de l'arbre tout en tenant compte du fait qu'à partir d'un certain nombre d'arbres dans le comité, l'amélioration de la performance devient négligeable [93]. D représente la profondeur maximum d'un arbre de décision. Limiter cette profondeur permet d'éviter le sur-entraînement de celui-ci. Cependant, une profondeur maximale trop faible peut dégrader la performance de l'arbre qui ne discrimine pas suffisamment les classes. Min représente le nombre de données minimum en entrée d'un nœud pour chercher la coupure optimale. Si la taille de l'ensemble des données en entrée est inférieure à Min alors le nœud devient une feuille et la classe du nœud est celle qui est majoritaire parmi les données en entrée. N' est la taille du sous-ensemble de données tiré aléatoirement pour la construction de chaque arbre de décision dans le comité. Enfin, F' est le nombre d'attributs tirés aléatoirement parmi les F attributs pour la construction de chaque noeud décisionnel dans l'arbre. L'ensemble contenant les F' attributs est noté A. L'ensemble des notations est illustré dans la figure 9.1 pour une meilleure compréhension.

9.1.1 Calcul de l'impureté basé sur le critère de Gini

Le calcul du critère de Gini permet de mesurer l'impureté d'une population de données. Une population est considérée pure lorsque toutes les données appartiennent à la

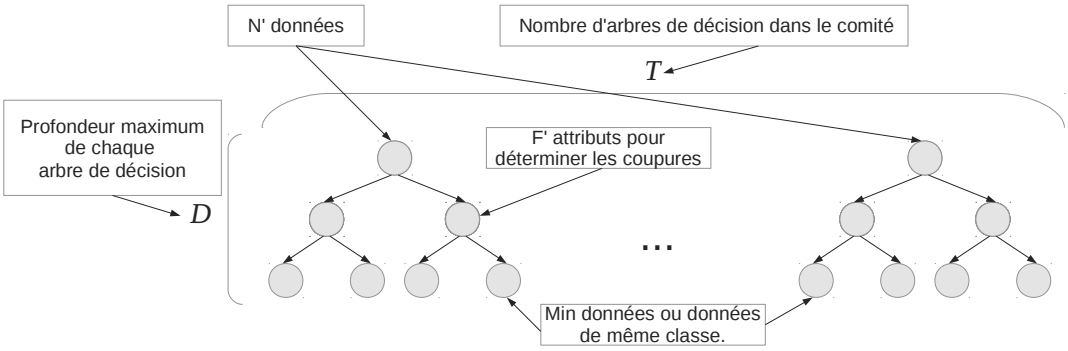


Figure 9.1 – Paramètres de l'algorithme forêt aléatoire.

même classe. Dans ce cas, l'impureté de la population est égale à 0 (équation 8.12). Lors de la construction d'un arbre de décision, nous sommes amenés à analyser des coupures qui correspondent au partitionnement d'une population de données E en 2 populations E_l et E_r. Mesurer l'évolution de l'impureté revient à calculer :

$$\Delta I(E, E_l, E_r) = Gini(E) - Gini(E_l) - Gini(E_r). \tag{9.1}$$

Si cette différence est importante, cela signifie que les populations E_l et E_r se rapprochent d'une population pure et c'est ce que nous recherchons. Ainsi, l'objectif est de maximiser l'équation 9.1, ce qui revient à minimiser $Gini(E_l) + Gini(E_r)$ car $Gini(E)$ est constant pour toutes les coupures possibles. Calculer $Gini(E_l)$ et $Gini(E_r)$ nécessite de connaître la fréquence d'apparition des classes dans chacune des populations (équation 8.12). Une structure de données, que nous nommons histogramme, est utilisée pour maintenir en mémoire le nombre de données par classe dans une population. L'histogramme est noté H_p avec $p \in \{l, r\}$ correspondant à chacune des populations E_l et E_r. Le nombre de données appartenant à une classe c est noté $H_p[c]$ par la suite. En supposant que n_p est la taille de la population E_p avec $p \in \{l, r\}$, le calcul de l'impureté de E_p est décrit par l'algorithme 9.1.

1: **procédure** GINI(H_p, n_p)
2: $\quad I = 0$
3: $\quad$ **pour** chaque classe c dans C **exécute**
4: $\quad\quad I = I + \frac{H_p[c]^2}{n_p^2}$
5: $\quad$ **fin pour**
6: $\quad I = 1 - I$
7: $\quad$ **retourne** $n_p \times I$
8: **fin procédure**

Algorithme 9.1 – Calcul de l'impureté de la population E_p.

La complexité en temps de cette opération est $O(K)$ car K itérations sont effectuées

pour considérer la fréquence d'apparition de chaque classe. La complexité en espace est $O(K)$ car il faut stocker l'histogramme contenant K entrées pour chaque classe.

9.1.2 Détermination de la coupure optimale

La détermination de la coupure optimale est effectuée en explorant F' attributs tirés aléatoirement parmi F. Le tirage aléatoire est effectué sans remplacement et chaque attribut a une probabilité égale d'être sélectionné.
Pour chaque attribut f dans A, la population E est triée suivant les valeurs de f dans un ordre croissant. Soit $E_f = \{v_1, \ldots, v_{n_p}\}$ les valeurs triées selon f. Il y a ainsi $n_p - 1$ coupures possibles et nous notons $E_l(i) = \{v_1, \ldots, v_i\}$ et $E_r(i) = \{v_{i+1}, \ldots, v_{n_p}\}$ une coupure avec $i \in [1, n_p - 1]$. Déterminer la coupure optimale consiste à sélectionner la coupure pour laquelle $Gini\,(E_l(i)) + Gini\,(E_r(i))$ est minimum. À partir de cette coupure optimale, nous construisons la règle de décision du nœud décisionnel courant qui s'exprime de la manière suivante :

$$x(f_{opt}) \leq S; \tag{9.2}$$

où $x(f_{opt})$ représente la valeur d'une donnée x suivant l'attribut de la coupure optimale f_{opt} et S représente la valeur seuil. Pour une coupure optimale $(E_l(i), E_r(i))$, S vaut v_i. L'ensemble de la procédure est détaillé dans l'algorithme 9.2.

Analysons la complexité en temps de cet algorithme. F' itérations sont effectuées (ligne 5) donc la complexité courante est $O(F')$. E est ensuite trié suivant l'attribut courant. La meilleure complexité en temps pour un algorithme de tri est $O(n \log n)$ [11]. La complexité courante est alors $O(F' \times n_p \log n_p)$. L'étape suivante consiste à explorer $n_p - 1$ coupures. Pour chaque coupure, nous calculons la différence d'impureté. La complexité courante est alors $O(F' \times n_p \log n_p + F' \times n_p \times K)$. Pour $n_p \to \infty$, la complexité en temps est dominée par le tri : $O(F' \times n_p \log n_p)$.
L'ensemble des données E et les histogrammes doivent être stockés en mémoire interne, la complexité en espace est alors $O(n_p + K)$.

9.1.3 Partitionnement des données et distribution aux nœuds fils

Une fois la coupure optimale déterminée, E doit être partitionné en 2 nouvelles populations E_l et E_r qui vont être distribuées respectivement aux nœuds fils gauche et droit. Le partitionnement s'effectue selon la règle de décision déterminée par la coupure optimale : $x(f_{opt}) \leq S$. Les données repectant cette règle sont ajoutées à E_l, sinon elles sont ajoutées à E_r. L'algorithme de partitionnement et de distribution est illustré par l'algorithme 9.3. La complexité en temps est $O(n_p)$ et la complexité en espace $O(n_p)$.

```
1: procédure CoupureOptimale(E, A)
2:     f_opt ← attribut de la coupure optimale.
3:     S ← seuil de la coupure optimale.
4:     ΔI_min ← impureté minimum.
5:     pour chaque attribut f ∈ A exécute
6:         E_f ← Trie E suivant f.
7:         H_l = ∅
8:         H_r ← distribution des classes dans E.
9:         n_l = 0
10:        n_r = n_p
11:        pour i = 1 ; i ≤ n_p − 1 ; i = i + 1 exécute
12:            H_l[y_i] = H_l[y_i] + 1
13:            H_r[y_i] = H_r[y_i] − 1
14:            n_l = n_l + 1
15:            n_r = n_r − 1
16:            si GINI(H_l, n_l) + GINI(H_r, n_r) < ΔI_min alors
17:                ΔI_min = GINI(H_l, n_l) + GINI(H_r, n_r)
18:                S = v_i
19:                f_opt = f
20:            fin si
21:        fin pour
22:    fin pour
23:    retourne (S, f_opt)
24: fin procédure
```

Algorithme 9.2 – Détermination de la coupure optimale.

```
1: procédure Partitionnement(E, E_l, E_r, f_opt, S)
2:     pour chaque x ∈ E exécute
3:         si x(f_opt) < S alors
4:             Ajout de x dans E_l.
5:         sinon
6:             Ajout de x dans E_r.
7:         fin si
8:     fin pour
9: fin procédure
```

Algorithme 9.3 – Partitionnement des données et distribution aux nœuds fils

9.1.4 Construction récursive de l'arbre de décision

La construction d'une arbre de décision est faite de manière récursive à partir du nœud racine. Soit B l'arbre de décision que nous voulons construire. B contient des nœuds décisionnels ou des nœuds feuilles qui sont identifiés par leur position dans l'arbre pos, un booléen indiquant si le nœud est une feuille $feuille$, la valeur de l'étiquette l si le nœud est une feuille, l'attribut de la coupure optimale f_{opt}, le seuil de la coupure optimal $seuil$, la profondeur du nœud dans l'arbre $profondeur$ et enfin les positions des nœuds fils $gpos$ (gauche) et $rpos$ (droit). Ainsi, le nœud à la position pos est $B[pos]$. Par exemple, l'accès à l'attribut $seuil$ d'un nœud à la position pos dans B se note $B[pos].seuil$. Avec ces notations, l'algorithme permettant de construire de manière récursive l'arbre est décrit dans l'algorithme 9.4.

L'algorithme récursif a plusieurs conditions d'arrêt. La première est remplie lorsque le nœud a atteint la profondeur maximum de l'arbre D. Le nœud devient alors une feuille de l'arbre et son étiquette correspond à la classe majoritaire dans E. La seconde condition est remplie lorsque le nombre de données dans E est inférieur à Min. De même, le nœud devient une feuille et son étiquette correspond à la classe majoritaire dans E. Enfin, la troisième condition d'arrêt est remplie lorsque toutes les données dans E appartiennent à la même classe C_k avec $k \in [1, K]$. Le nœud devient une feuille et l'étiquette prend la valeur C_k.

Si aucune condition d'arrêt n'est vérifiée alors la prochaine étape consiste à déterminer la coupure optimale à partir d'un ensemble A composé de F' attributs tirés aléatoirement (ligne 17). La coupure optimale est déterminée et retourne une paire de valeurs correspondant à l'attribut optimal f_{opt} et au seuil S qui sont affectées aux attributs du nœud courant (lignes 19 et 20). L'étape suivante consiste à ajouter deux nœuds fils au nœud courant : le nœud fils gauche à la position $gpos$ et le nœud fils droit à la position $rpos$. Ces nouveaux nœuds sont ajoutés à la fin de B et leur profondeur est incrémentée par rapport à la profondeur du nœud courant (lignes 23 et 24). Le partitionnement (ligne 26) établit les nouvelles populations E_l et E_r à distribuer aux nœuds fils gauche et droit respectivement. Enfin, la procédure continue récursivement avec la construction des nœuds fils gauche et droit (lignes 27 et 28).

Analysons la complexité en temps de l'algorithme. Nous considérons qu'à chaque niveau de l'arbre, les n_p données sont distribuées parmi les nœuds de l'arbre. Pour un niveau donné, le coût de construction d'un nœud sur une population de taille n est dominé par la détermination de la coupure optimale dont la complexité en temps est $O(F' \times n \log n)$. La somme des tailles des populations en entrée des nœuds d'un même niveau est égale à n_p. Ainsi la complexité en temps pour construire un niveau de l'arbre est $O(F' \times n_p \log n_p)$. Sachant que la profondeur maximum de l'arbre est D, il résulte que la complexité en temps pour la construction d'un arbre de décision est $O(D \times F' \times n_p \log n_p)$.

Analysons la complexité en espace. La population initiale de taille n_p doit être maintenue en mémoire interne. De plus, pour la détermination des coupures optimales, deux histogrammes contenant K classes sont utilisés. La complexité en espace est $O(n_p + 2 \times$

1: **procédure** CONSTRUCTIONARBRE(E, B, pos)
2: **si** $B[pos].profondeur == D$ **alors**
3: $B[pos].feuille = vrai$
4: $B[pos].l =$ classe majoritaire dans E.
5: Arrête la récursion.
6: **fin si**
7: **si** taille(E)$< Min$ **alors**
8: $B[pos].feuille = vrai$
9: $B[pos].l =$ classe majoritaire dans E.
10: Arrête la récursion.
11: **fin si**
12: **si** les données dans E sont de la même classe C_k **alors**
13: $B[pos].feuille = vrai$
14: $B[pos].l = C_k$
15: Arrête la récursion.
16: **fin si**
17: $A \leftarrow$ tire F' attributs aléatoirement parmi F attributs.
18: $(S, f_{opt}) \leftarrow$ CoupureOptimale(E, A)
19: $B[pos].seuil = S$
20: $B[pos].f_{opt} = f_{opt}$
21: $B[pos].gpos =$ taille(B)
22: $B[pos].rpos =$ taille(B) +1
23: $B[gpos].profondeur = B[pos].profondeur + 1$
24: $B[rpos].profondeur = B[pos].profondeur + 1$
25: $(E_l, E_r) \leftarrow$ nouvelles populations.
26: Partitionnement(E, E_l, E_r, f_{opt}, S)
27: ConstructionArbre(E_l, B, $B[pos].gpos$)
28: ConstructionArbre(E_r, B, $B[pos].rpos$)
29: **fin procédure**

Algorithme 9.4 – Construction récursive de l'arbre

K).

9.1.5 Construction de la forêt aléatoire

La forêt aléatoire est un ensemble de T arbres de décision où chaque arbre est construit à partir d'un sous-ensemble de données E' tiré aléatoirement à partir de l'ensemble des données initial E. L'algorithme 9.5 décrit la procédure.

La complexité en temps de l'algorithme est $O(T \times D \times F' \times n_p \log n_p)$ et la complexité en espace est $O(n_p + 2 \times K + 2^{D+1})$ car il faut stocker les noeuds de l'arbre. En revanche,

1: **procédure** ConstructionForêtAléatoire(T, E)
2: **pour** $t = 0$; $t < T$; $t = t + 1$ **exécute**
3: $B \leftarrow$ nouvel arbre de décision.
4: $E' \leftarrow$ sous-ensemble tiré aléatoirement à partir de E.
5: ConstructionArbre(E', B, 0)
6: **fin pour**
7: **fin procédure**

Algorithme 9.5 – Calcul de l'impureté de la population E_p.

nous considérons, qu'après la construction de chaque arbre de décision, celui-ci est stocké dans la mémoire externe.

9.2 Passage à l'échelle : état de l'art

Dans cette section, nous établissons une revue des principales solutions proposées dans la littérature pour le passage à l'échelle des algorithmes basés sur la construction d'arbres de décision dans la situation où les données d'apprentissage ne peuvent être stockées dans la mémoire interne de l'ordinateur.

9.2.1 La méthode SLIQ

Dans la section précédente, nous avons vu que la construction de chaque nœud requiert de trier la population des données en entrée suivant les attributs contenus dans A. Par conséquent, cela nécessite d'effectuer F' tris pour chaque nœud et donc $(2^D - 1) \times F'$ tris en moyenne pour la construction de chaque arbre. Dans [98], les auteurs proposent la méthode SLIQ ("Supervised Learning In Quest") pour la construction d'un arbre de décision. Dans cette méthode, ils suppriment l'opération redondante du tri en ne triant qu'une seule fois les données pour chaque attribut avant la construction de l'arbre. Pour cela, la population de données intiale est décomposée en F listes d'attributs et une liste pour les étiquettes des données. Chaque liste d'attributs est composée de paires contenant la valeur de la donnée suivant l'attribut ainsi que la position de celle-ci dans la liste des étiquettes :

$$\begin{bmatrix} L_{a1} = \{[x_1(1), 0], \ldots, [x_n(1), n-1]\} \\ L_{a2} = \{[x_1(2), 0], \ldots, [x_n(2), n-1]\} \\ \cdot \\ \cdot \\ \cdot \\ L_{aF} = \{[x_1(F), 0], \ldots, [x_n(F), n-1]\} \end{bmatrix} \tag{9.3}$$

La liste des étiquettes est composée de paires contenant l'étiquette de la donnée correspondante et la position du nœud dans l'arbre de décision où se situe la donnée (au début

les données sont situées au nœud racine de l'arbre) :

$$L_c = \{[y_1, 0], \ldots, [y_n, 0]\} \tag{9.4}$$

Les auteurs supposent que l'ordinateur a une capacité de mémoire interne suffisante pour stocker la liste des étiquettes. Par contre, les listes d'attributs peuvent être stockées dans la mémoire externe si nécessaire. La première étape avant de construire l'arbre consiste à trier chaque liste d'attributs par ordre croissant. Aucun détail n'est cependant donné dans la situation où la liste d'attributs ne peut être contenue en mémoire interne.
La construction de l'arbre de décision est effectuée de manière transversale où les coupures optimales des nœuds d'un même niveau de l'arbre sont déterminées en ne visitant qu'une seule fois chaque liste d'attributs. Dans le contexte de leurs travaux, les auteurs n'introduisent pas d'aléa pour les attributs. Ainsi, pour chaque nœud la coupure optimale est déterminée en visitant toutes les listes d'attributs. La détermination des coupures optimales pour un niveau de l'arbre est illustrée par l'algorithme 9.6.

1: **procédure** CoupuresOptimales
2: **pour** chaque $finF$ **exécute**
3: Visite la liste L_{af}.
4: **pour** chaque valeur $v \in L_{af}$ **exécute**
5: $pos_c \leftarrow$ position dans L_c
6: $pos_n \leftarrow$ position dans l'arbre.
7: Met à jour H_l et H_r correspondant à $B[pos_n]$
8: Calcule la différence d'impureté.
9: **fin pour**
10: **fin pour**
11: **fin procédure**

Algorithme 9.6 – Détermination des coupures optimales pour tous les nœuds d'un niveau de l'arbre proposée dans [98]

Certains détails ont été omis dans la description de l'algorithme 9.6. En particulier, pour chaque nœud d'un même niveau de l'arbre, deux histogrammes sont alloués pour calculer la différence d'impureté. Ainsi pour chaque nouvelle valeur dans une liste d'attributs, les histogrammes sont mis à jour et la différence d'impureté est calculée. Si celle-ci est minimum, alors elle est gardée en mémoire pour le nœud ainsi que le seuil et l'attribut optimal correspondants. Après avoir visité toutes les listes d'attributs, sont connues les coupures optimales pour chaque nœud.
L'étape suivante de SLIQ effectue la partition des données en créant les nœuds fils des nœuds du niveau courant de l'arbre. Une fois les attributs déterminés pour la coupure de chaque noeud, l'étape suivante consiste à visiter les listes de ces attributs pour distribuer les données aux noeuds fils. Pour chaque paire de valeurs dans la liste d'attributs, nous récupérons la position du nœud dans L_c et nous déterminons si la donnée correspondante doit être distribuée au nœud fils gauche ou au nœud fils droit selon la règle de décision.

L'entrée dans L_c correspondante, est ensuite mise à jour avec la nouvelle position du nœud dans l'arbre. L'algorithme est illustré par l'algorithme 9.7.

1: **procédure** PARTIONNEMENT
2: **pour** chaque attribut f utilisé dans une coupure **exécute**
3: Visite la liste L_{af}.
4: **pour** chaque valeur $v \in L_{af}$ **exécute**
5: $pos_c \leftarrow$ position dans L_c
6: $pos_n \leftarrow$ position dans l'arbre.
7: **si** v vérifie la règle de décision **alors**
8: Distribution au nœud fils gauche g de pos_n
9: **sinon**
10: Distribution au nœud fils droit r de pos_n.
11: **fin si**
12: Mise à jour de L_c.
13: **fin pour**
14: **fin pour**
15: **fin procédure**

Algorithme 9.7 – Partitionnement et distribution des données aux nœuds fils.

Les auteurs proposent de supprimer, à chaque itération, les valeurs dans les listes d'attributs qui correspondent à des données appartenant à des nœuds feuille. Cela permet d'avoir des listes d'attributs qui se réduisent au fur et à mesure de la construction des niveaux de l'arbre. Cependant, il faut garder à l'esprit qu'écrire de nouvelles listes d'attributs dans la mémoire externe à un coût au niveau du temps d'exécution qui est loin d'être négligeable.

Le principal inconvénient de cette méthode est qu'il est nécessaire de stocker L_c en mémoire tout au long de la procédure. Cette solution n'est donc pas échelonnable. De plus, l'écriture de nouvelles listes d'attributs dans la mémoire externe peut augmenter le temps d'exécution de manière conséquente pouvant rendre inutilisable cette méthode sur des grands volumes de données d'apprentissage.

9.2.2 La méthode SPRINT

Dans [99], les auteurs identifient un inconvénient majeur dans la méthode SLIQ. En effet, nous venons de voir qu'elle impose que la liste des étiquettes soit maintenue en mémoire interne tout au long de la procédure. Cet algorithme n'est donc pas échelonnable. Par conséquent, les auteurs proposent un nouvel algorithme nommé SPRINT ("Scalable PaRallelizable INduction of decision Trees") utilisant des structures différentes de celles utilisées pour SLIQ. L'objectif est de garder la propriété intéressante de SLIQ qui consiste à ne trier qu'une seule fois les listes d'attributs avant la construction de l'arbre

de décision et d'éviter de maintenir L_c en mémoire interne. Pour cela, les listes d'attributs sont toujours utilisées, mais elles contiennent de l'information supplémentaire : la valeur de l'attribut de la donnée correspondante, l'étiquette de la donnée ainsi que la position de la donnée dans l'ensemble des données intiales.
Ces listes sont ainsi triées au début de la procédure et sont écrites dans la mémoire externe si nécessaire. La différence avec SLIQ intervient lorsque les données doivent être distribuées aux nœuds fils des nœuds du niveau courant après avoir déterminé les coupures optimales. Lors du partionnement, sont créées pour chaque nœud du niveau courant de l'arbre, 2 nouvelles listes d'attributs pour chaque nœud fils. Nous notons au passage que cette opération d'écriture est coûteuse en temps s'il est nécessaire d'écrire dans la mémoire externe. La première étape consiste à parcourir les listes d'attributs sélectionnées pour les coupures et à créer les nouvelles listes. Cependant, il est aussi nécessaire de partitionner les listes d'attributs qui n'ont pas été sélectionnées pour les coupures. Pour cela, les auteurs proposent d'utiliser une table de hachage qui associe la position de la donnée dans l'ensemble à la position du nœud où la donnée a été déplacée. Ainsi, les listes d'attributs restantes sont visitées pour créer les nouvelles listes. Si la table de hachage est trop volumineuse pour être contenue en mémoire, alors le partitionnement est effectué en plusieurs fois. Cependant, aucun détail n'est donné sur la façon de procéder.

9.2.3 La méthode RainForest

Dans [100], les auteurs proposent d'améliorer la performance de l'algorithme SPRINT au prix de faire apparaître de nouveau la contrainte mémoire. En effet, ils proposent de revenir à l'algorithme de référence pour la construction d'un arbre de décision en triant les données suivant les attributs pour la construction de chaque nouveau nœud. L'idée consiste à utiliser pour un attribut f une représentation réduite des données qu'ils nomment des AVC-sets ("Attribute-Value, Class-Label"). Cette nouvelle représentation consiste à regrouper les données ayant la même valeur selon f avec un histogramme représentant la distribution des classes de ces données. Ainsi, la taille d'un AVC-set dépend du nombre de valeurs distinctes suivant un attribut. Pour trouver la meilleure coupure selon un attribut, l'opération consiste à manipuler une liste de m AVC-sets et d'analyser les $m-1$ coupures possibles. Un inconvénient de cette méthode est lorsque le nombre de valeurs distinctes suivant un attribut est élevé. Dans cette situation, stocker les AVC-sets revient à stocker l'ensemble des données d'apprentissage. Si l'ensemble des AVC-sets ne peut être stocké en mémoire interne, alors la liste des AVC-sets doit être explorée plusieurs fois pour déterminer les différences d'impureté pour chaque coupure.

Afin de pallier ce problème, les auteurs dans [101] proposent de combiner cette approche avec une technique d'échantillonnage. Leur solution consiste à partitionner les attributs pour lesquels il y a un grand nombre de valeurs distinctes en intervalles de largeurs égales qui correspondront à des AVC-sets. Cela signifie que chaque intervalle contient le même nombre de valeurs. Le nombre d'intervalles est paramétré par l'utilisateur et influence significativement la performance du classifieur. L'inconvénient de cette

méthode est qu'elle ne garantit pas des arbres de décision identiques à ceux obtenus si l'ensemble des données pouvait être stocké en mémoire.

L'échantillonnage est aussi utilisé dans [102] où les auteurs indiquent qu'il est très coûteux en temps de calcul de construire des arbres de décision à partir de larges bases de données d'apprentissage où les attributs sont représentés par des valeurs numériques continues. Les auteurs présentent ainsi une solution qui permet d'échantillonner l'ensemble des données initial tout en minimisant la dégradation de la performance du classifieur. L'inconvénient ici pour notre investigation est le fait que cette solution n'assure pas toujours un classifieur identique à celui obtenu à partir de l'ensemble des données d'apprentissage.

9.2.4 Google PLANET

Google PLANET ("Parallel Learner for Assembling Numerous Ensemble Trees") [103] est une méthode qui permet de déployer la construction d'un arbre de décision en utilisant le modèle de programmation MapReduce proposé dans [104] sur un environnement parallèle et distribué. Le modèle de programmation MapReduce est souvent appliqué dans un environnement distribué comme par exemple un cluster composé de plusieurs ordinateurs pouvant communiquer à travers le réseau. Ce modèle est composé de 2 étapes : Map et Reduce. Lors de l'étape Map, chaque nœud du cluster applique une fonction $\text{Map}(k, v)$ sur une paire (k, v) stockée dans sa mémoire interne locale. k est une clé identifiant une donnée v sur laquelle la fonction Map définie par l'utilisateur est appliquée. Une fonction $\text{Map}(k, v)$ est appelée pour chaque paire (k, v). La fonction $\text{Map}(k, v)$ retourne une nouvelle paire (k', v'). La seconde étape est la fonction Reduce, qui dans un premier temps regroupe toutes les valeurs v' qui ont la même clé k', et qui dans un second temps réduit la paire $(k', v'_1, ..., v'_n)$ pour fournir (k', v''). Une fonction Reduce est appelée pour chaque clé unique k'.

L'algorithme Google PLANET est une adaptation du modèle MapReduce pour la construction d'un arbre de décision. L'arbre de décision proposé est un modèle de régression qui utilise comme mesure d'impureté la variance notée Var :

$$\Delta I(E_l, E_r) = |E| \times Var(E) - (|E_l| \times Var(E_l) + |E_r| \times Var(E_r)) \tag{9.5}$$

Google PLANET combine plusieurs stratégies d'exécution lors de la procédure suivant si les données en entrée d'un nœud de l'arbre de décision peuvent être stockées en mémoire interne ou pas. La gestion de ces différentes stratégies est effectuée par le contôleur qui est un nœud du cluster chargé de plannifier et contrôler l'ensemble de la procédure de construction. Il maintient aussi dans sa mémoire interne l'arbre de décision en cours de construction. Ainsi, à tout moment au cours de la procédure, le contrôleur connaît la partition des données dans l'arbre et détermine les nœuds pour lesquels la coupure optimale doit être déterminée. Supposons un nœud qui doit être construit, le contrôleur a 2 choix possibles concernant la stratégie à plannifier pour déterminer la coupure optimale. Si l'ensemble des données en entrée du nœud peut être stocké dans la mémoire interne du nœud du cluster, alors la stratégie $InMemory$ est sélectionnée et la construction

du sous-arbre engendré par le nœud est achevée sur ce nœud du cluster. À l'inverse, si l'ensemble des données ne peut être contenu en mémoire alors la coupure optimale est déterminée en utilisant plusieurs nœuds du cluster (stratégie $MapReduce$).

Au lieu de maintenir des listes d'attributs triées comme dans SLIQ et SPRINT, Google PLANET utilise des histogrammes de profondeur égale [105]. Étant donné une liste de valeurs numériques triée $\{1, \ldots, N\}$, l'élement situé à la position $\lceil \phi N \rceil$ où $\phi \in [0, 1]$ est le ϕ-quantile. Les histogrammes de profondeur égale sont alors les $\frac{i}{p}$-quantiles avec $i \in \{1, \ldots, p-1\}$ et p une valeur choisie judicieusement. Ainsi, chaque $\frac{i}{p}$-quantile représente une coupure possible. La constitution de l'ensemble des coupures à analyser est effectuée avant l'étape de construction de l'arbre de décision et est décrite par la suite.
La première étape est une fonction Map qui consiste à charger l'ensemble des coupures à analyser pour chaque attribut. Pour chaque nœud du cluster, une table de hachage est maintenue par le contrôleur où les clés sont les coupures à considérer et les valeurs un tuple contenant les informations statistiques suffisantes pour calculer la différence d'impureté à savoir $\{\sum v, \sum v^2, \sum 1\}$. Pour une coupure située à la position s, la première expression représente la somme des valeurs des données en entrée du nœud qui sont inférieures à s, la deuxième la somme du carré des valeurs qui sont inférieures à s et la troisième expression le nombre de données dont les valeurs sont inférieures à s. Ces tuples sont construits par la fonction Map qui visite les valeurs des données en entrée. La sortie de chaque fonction Map est une paire clé/valeur où la clé est la position de la coupure et la valeur un tuple. Une fonction Reduce regroupe ensuite les sorties des fonctions Map par clé pour constituer les tuples résultants. Pour déterminer la coupure optimale, les valeurs des données doivent être triées par ordre croissant. Or, Google PLANET ne considère pas toutes les valeurs des données mais seulement les coupures qui sont les $\frac{i}{p}$-quantiles. La fonction Reduce est aussi chargée de trier ces coupures par ordre croissant.
Le second rôle de la fonction Reduce est de trouver la coupure locale optimale parmi les coupures qu'elle reçoit suivant un attribut f. Supposons la coupure s pour un nœud n de l'arbre de décision et un attribut f. La fonction Reduce regroupe l'ensemble des données dont les valeurs suivant f sont inférieures à s, noté $E_l = \{\sum v_l, \sum v_l^2, \sum 1\}$, l'ensemble des données dont les valeurs sont supérieures à s, noté $E_r = \{\sum v_r, \sum v_r^2, \sum 1\}$ et calcule la différence d'impureté $\Delta I(E_l, E_r)$ avec l'équation 9.5. Finalement, chaque fonction Reduce fournit la coupure optimale qu'elle a vue pour chaque nœud de l'arbre et le contrôleur détermine la coupure optimale pour chaque nœud de l'arbre de décision. Le contrôleur met ensuite à jour les nouveaux nœuds de l'arbre à construire et la procédure continue.

Les auteurs ont identifié plusieurs problèmes avec la solution proposée. La construction de chaque nouveau nœud nécessite de lancer un nouveau MapReduce. Or, il peut y avoir de nombreux nœuds à construire surtout si l'ensemble des données au départ est grand. Par conséquent, de nombreuses opérations de communication sont nécessaires pour trouver à chaque fois les coupures à analyser, trier les coupures et déterminer les coupures optimales augmentant ainsi le temps d'exécution. En effet, à l'origine le modèle

d'exécution MapReduce n'est pas conçu pour des algorithmes qui requièrent beaucoup d'itérations à moins qu'ils ne soient facilement parrallélisables. De plus, le temps d'exécution pour visiter les attributs pour déterminer les coupures à analyser s'est avéré important, mettant en question le gain obtenu avec la parallélisation dans certaines situations.

9.2.5 Forêt aléatoire utilisant MapReduce

Plus récemment dans [106], les auteurs proposent d'implémenter l'algorithme des forêts aléatoires en utilisant le modèle de programmation MapReduce sur un environnement distribué. T sous-ensembles de données d'apprentissage sont constitués aléatoirement pour la construction de T arbres de décision. Chaque sous-ensemble est distribué à un nœud du cluster. La fonction Map construit à partir d'un sous-ensemble de données un arbre de décision en utilisant la mesure d'impureté proposée dans [91] pour déterminer la coupure optimale de chaque nœud. Chaque nœud du cluster utilisé pour la fonction Map, contient un modèle de l'arbre de décision. La fonction Reduce est utilisée pour effectuer la prédiction par un vote majoritaire du comité d'arbres de décision.
Le principal inconvénient de cette méthode, qui n'est pas mentionné par les auteurs, est que la construction de chaque arbre de décision implique que le sous-ensemble des données à partir duquel il est construit puisse être contenu en mémoire. Effectivement, il sera toujours possible de créer des ensembles pouvant être stockés en mémoire mais cela a une limite. En effet, si nous supposons que l'ensemble des données initial occupe des téraoctets de données et que les nœuds du cluster ont une mémoire interne de quelques gigaoctets alors chaque sous-ensemble de données représente un millième de l'ensemble des données initial. Dans ce cas, chaque sous-ensemble peut ne plus être représentatif et le comité d'arbres de décision construit à partir de ces sous-ensembles peut avoir une mauvaise performance de prédiction.

9.2.6 Discussions

Nous venons d'établir une revue des différentes techniques existantes pour le passage à l'échelle de la construction d'un ensemble d'arbres de décision. Les méthodes SLIQ et SPRINT proposent une solution exacte pour la construction d'un arbre de décision mais au prix de nombreuses écritures dans la mémoire externe augmentant de manière significative le temps d'exécution. Ces solutions peuvent ainsi s'avérer inutilisables pour le traitement de grands ensembles de données d'apprentissage. La méthode RainForest propose de réduire la taille des données en regroupant les données dont les valeurs suivant un attribut sont égales. Cette solution fonctionnant pour des attributs catégoriques peut s'avérer inefficace dans le cas où les attributs des données sont numériques avec un grand nombre de valeurs distinctes. Google PLANET propose une solution intéressante en combinant plusieurs stratégies pour construire les nœuds dans un arbre de décision suivant la taille des données en entrée. En revanche, la construction de chaque nœud implique un nouveau MapReduce et une nouvelle exploration des données d'apprentissage

pour établir les coupures qui vont être utilisées pour déterminer la coupure optimale. Ces opérations s'avèrent coûteuses principalement à cause de la communication engendrée pour la distribution des données parmi les nœuds du cluster. Enfin, contraindre la taille de l'ensemble des données suivant la quantité de mémoire interne disponible pour construire un arbre de décision peut sévèrement dégrader la performance du classifieur final surtout si l'ensemble des données d'apprentissage initial est massif.

Ayant analysé ces méthodes et leurs inconvénients, nous proposons un algorithme baptisé SMART ("Scalable Multi strAtegy Random Trees") qui associe l'avantage de n'effectuer qu'un seul tri pour la construction du comité d'arbres de décision tout en minimisant l'écriture dans la mémoire externe par l'utilisation de plusieurs stratégies de construction de nœuds. La première contribution est le fait de ne trier qu'une seule fois pour la construction de tous les arbres du comité. Cette contribution est inspirée des méthodes SLIQ et SPRINT. La deuxième contribution est l'utilisation de plusieurs stratégies pour déterminer les coupures optimales des nœuds lors de la construction d'un arbre. Elle a pour rôle de minimiser les écritures dans la mémoire externe et est inspirée par la méthode proposée dans Google PLANET. Nous montrons que notre algorithme ne dépend pas de la taille des données et permet d'obtenir un classifieur identique à celui que nous aurions obtenu si l'ensemble des données d'apprentissage pouvait être contenu en mémoire interne. Pour le prouver, nous introduisons une méthode qui consiste à comparer 2 arbres de décision. Enfin, notre algorithme sera testé sur un volume de données d'apprentissage issu de séries multitemporelles recouvrant toute la zone sud de la France.

Chapitre 10

Algorithme SMART

10.1 Aperçu général de l'algorithme SMART

Dans la suite, nous considérons un ensemble de données d'apprentissage $E = \{x_i, y_i\}_{i=1}^m$ avec $(x_i, y_i) \in \mathbb{R}^F \times \mathbb{N}^*$. T représente le nombre d'arbres de décision à construire et n le nombre de données dans le sous-ensemble E' tiré aléatoirement pour la construction de chaque arbre de décision. S représente une stratégie pour construire un noeud suivant la taille des données en entrée. Enfin, la mémoire interne disponible est notée M_R et est exprimée en octets. Dans la suite, nous nous basons sur le diagramme d'exécution illustré par la figure 10.1 dont nous expliquons les étapes principales.

10.1.1 Génération des listes d'attributs

Comme pour les méthodes SLIQ et SPRINT, la première étape consiste à générer les listes d'attributs. Si $\mathbb{R}^F$ est l'espace des attributs, cela revient à construire F listes contenant m valeurs. Par la suite, nous notons $x_i(f)$ la valeur d'une donnée x_i suivant l'attribut f. Les méthodes SLIQ et SPRINT permettent la construction d'un seul arbre de décision. Dans notre situation, nous devons construire un comité de T arbres de décision en introduisant de l'aléa dans les sous-ensembles pour chaque arbre et dans l'ensemble des attributs pour la construction de chaque noeud décisionnel de chaque arbre. L'objectif ici est d'étendre l'avantage de ne trier qu'une seule fois les listes d'attributs pour la construction de l'ensemble des arbres. Pour ce faire, nous allons affecter à chaque donnée x_i une séquence de T bits issue d'un tirage aléatoire effectué avec une distribution de Bernouilli en indiquant le nombre de bits à 1 que nous voulons dans la séquence. Cette séquence de bits permet de construire les sous-ensembles de données d'apprentissage pour chaque arbre de décision. Dans notre cas, nous voulons qu'il y ait n bits à 1 parmi m bits. Supposons que $T = 8$ et que $n = \lceil \frac{3m}{4} \rceil$, c'est-à-dire la partie entière supérieure. Cela signifie que le sous-ensemble de données pour chaque arbre représente 75% de l'ensemble de données initial. Autrement dit, chaque donnée d'apprentissage a 3 chances sur 4 d'être considérée pour la construction d'un arbre. Un exemple de séquence

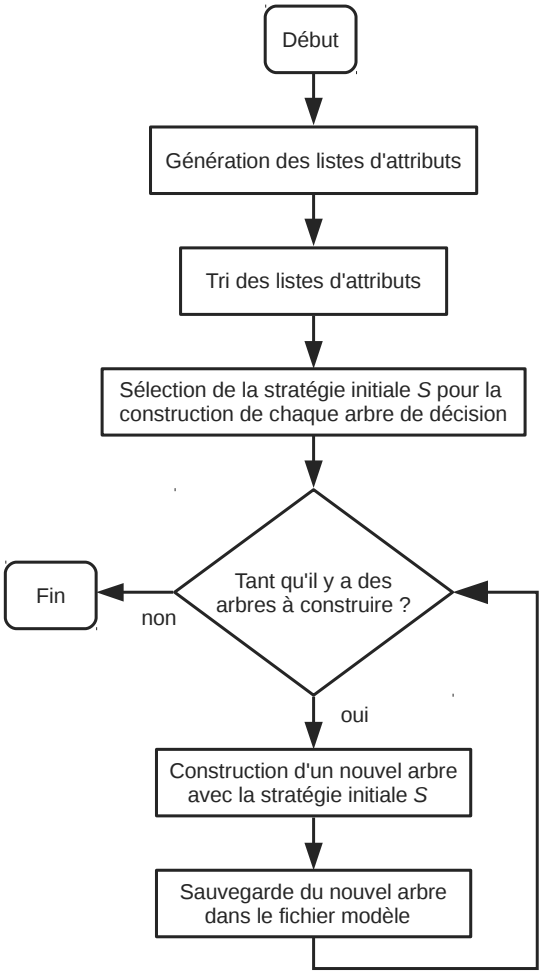


Figure 10.1 – Diagramme d'exécution de l'algorithme SMART

de bits pour une donnée serait [10110111], ce qui signifie qu'elle est considérée pour la construction du premier arbre, du troisième arbre, *etc.*

Dans une liste d'attributs, un noeud contient plusieurs informations similaires à celles proposées pour la méthode SPRINT. En supposant f l'attribut correspondant, un noeud contient l'étiquette y_i de la donnée, la position i dans E, la valeur de x_i suivant f notée $x_i(f)$ et enfin la séquence de T bits notée b_i.

10.1.2 Tri des listes d'attributs

Après avoir généré les listes d'attributs, l'étape suivante consiste à les trier. Il faut considérer la situation où la liste d'attributs est très volumineuse et ne peut être stockée en mémoire interne. Nous avons vu dans les sections 1.5.1 et 1.5.2, 2 stratégies pour trier des données ne pouvant être stockées en mémoire interne. L'algorithme de tri par

flot de données requiert un nombre d'accès important dans la mémoire externe puisque sa complexité est quadratique, tandis que l'algorithme External Merge Sort a une complexité $O(n \log n)$ et requiert ainsi beaucoup moins d'accès dans la mémoire externe ($O(\log n)$ accès). Par conséquent, dans la situation où les listes d'attributs ne peuvent être contenues en mémoire interne, nous choisissons d'utiliser l'algorithme de tri External Merge Sort dont le fonctionnement est décrit dans la figure 1.7. La librairie STXXL [18] propose une implémentation générique de cet algorithme avec la même interface que l'algorithme de tri proposé dans la STL[13]. L'utilisation de l'algorithme nécessite de connaître la quantité de mémoire interne disponible pour déterminer la taille des blocs de données initiaux. Une fois triées, les listes d'attributs sont ensuite stockées dans la mémoire externe pour la construction de l'ensemble des arbres de décision.

10.1.3 Choix de la stratégie initiale

Au départ de la construction d'un arbre de décision, il existe 3 situations possibles suivant la taille de E' :

1. L'ensemble des listes d'attributs peut être stocké en mémoire interne. La stratégie "En Mémoire", notée EM, est sélectionnée pour contruire le noeud racine de l'arbre. Cette stratégie consiste tout simplement à appliquer un algorithme similaire à l'algorithme de référence tel que décrit dans le chapitre 9.

2. L'ensemble des listes d'attributs ne peut être stocké en mémoire. Cependant, le partitionnement des données dans l'arbre peut être stocké en mémoire interne. La stratégie "Partitionnement En Mémoire", notée PEM, est sélectionnée pour construire le noeud racine de l'arbre. Cette stratégie est similaire à celle proposée par la méthode SPRINT car elle utilise une table de hachage contenant la localisation de chaque donnée dans l'arbre de décision.

3. L'ensemble des listes d'attributs ainsi que le partitionnement des données ne peuvent être stockés en mémoire interne. La stratégie "Hors Mémoire", notée HM par la suite, consiste à utiliser une table de hachage par morceaux contenant le partitionnement des données dans l'arbre. Une solution pour la représentation de cette table de hachage en mémoire externe est proposée.

Nous établissons ainsi une hiérarchie au niveau des stratégies. A tout moment durant la procédure, un changement de stratégie d'exécution peut s'opérer si les conditions pour le changement sont vérifiées. Ainsi, lors de l'exécution de la stratégie HM, nous pouvons changer en choisissant soit la stratégie PEM soit la stratégie EM et lors de l'exécution de la stratégie PEM nous pouvons changer et choisir la stratégie EM. La hiérarchie des stratégies d'exécution est illustrée par la figure 10.2. L'objectif est de concevoir un algorithme capable de s'adapter intelligemment à la taille des données à traiter lors de l'exécution. Contrairement aux méthodes SLIQ et SPRINT, nous minimisons l'écriture en mémoire externe, ce qui va permettre d'obtenir de meilleures performances concernant

le temps d'exécution et ainsi rendre l'algorithme forêt aléatoire réellement échelonnable pour l'exploitation de volumes de données d'apprentissage de taille arbitraire.

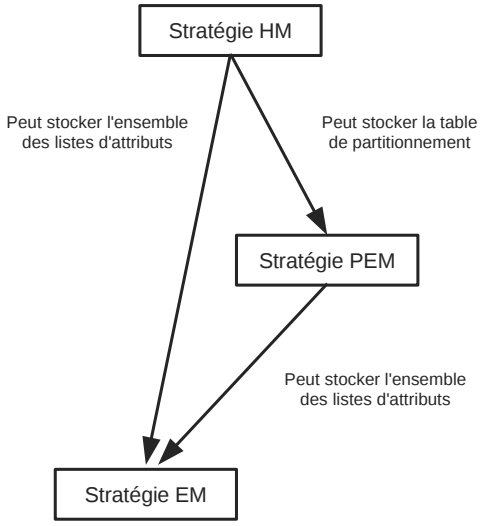


Figure 10.2 – Hiérarchie des stratégies de l'algorithme SMART

10.2 Description des stratégies d'exécution dans SMART

Nous présentons dans cette section les différentes stratégies d'exécution utilisées dans SMART et la façon dont celles-ci sont sélectionnées suivant la taille des données à traiter.

10.2.1 Stratégie "En Mémoire"

Étape d'initialisation

Si au cours de l'exécution d'une stratégie PEM ou HM, nous identifions que l'ensemble des données en entrée d'un nouveau nœud décisionnel a une taille qui permet de passer à la stratégie EM alors le relais vers cette stratégie est effectué. Le protocole permettant d'effectuer ce relais est conçu pour réaliser le moins d'écritures possibles dans la mémoire externe. En effet, nous avons vu que le principal inconvénient des méthodes SLIQ, SPRINT mais aussi Google PLANET est l'écriture des nouvelles populations de données issues d'un partitionnement dans la mémoire externe. L'objectif est de minimiser ces écritures car cette opération est très lente. Lorsqu'une population de données E' peut être stockée en mémoire interne, la stratégie PEM ou HM écrit un fichier dans la mémoire externe contenant seulement les positions des données dans E. Ce fichier est

noté I_f. Par exemple, si nous supposons les listes d'attributs triées initiales de E :

$$\begin{bmatrix} x_5(1) & x_7(1) & x_3(1) & x_1(1) & x_8(1) & x_2(1) & x_6(1) & x_4(1) & x_0(1) \\ x_0(2) & x_6(2) & x_4(2) & x_2(2) & x_8(2) & x_3(2) & x_1(2) & x_5(2) & x_7(2) \\ x_7(3) & x_2(3) & x_5(3) & x_1(3) & x_3(3) & x_0(3) & x_4(3) & x_8(3) & x_6(3) \end{bmatrix} \tag{10.1}$$

Nous supposons les données x_5, x_7, x_3 en entrée d'un nouveau nœud sachant que nous pouvons stocker seulement 3 données en mémoire interne. La stratégie va alors écrire un fichier contenant les positions de ces données :

$$I_f = [5, 7, 3]$$

Dans un premier temps, nous construisons l'ensemble des données d'apprentissage qui va être stocké en mémoire interne. Pour ce faire, 2 structures de données sont utilisées. La première est une table dynamique de taille F qui contient les listes d'attributs représentées également sous la forme de tables dynamiques. Les listes d'attributs sont allégées cr l'information à stocker pour chaque donnée est réduite. En effet, chaque entrée dans une liste pour l'attribut f contient 2 valeurs : $x_i(f)$ et la position i de x_i dans E. La seconde structure de données est une table de hachage contenant des paires clé/valeur où la clé est la position de la donnée d'apprentissage et la valeur est l'étiquette de la donnée. Ces 2 structures de données sont initialisées à partir des listes d'attributs triées de E stockées en mémoire externe et du fichier I_f créé par la stratégie supérieure. Nous notons *Attr* la table contenant les listes d'attributs triées pour les données en entrée du nœud, L la table de hachage contenant les étiquettes et A_f les fichiers contenant les listes d'attributs triées de E. Avec ces notations, l'algorithme d'initialisation est illustré par l'algorithme 10.1.

En reprenant notre exemple précédent, à l'issue de la ligne 3 nous obtenons la table de hachage suivante : $L = [3 \to 0, 5 \to 0, 7 \to 0]$. Après le parcours des listes d'attributs (ligne 5 à ligne 14) nous obtenons :

$$Attr = \begin{bmatrix} x_5(1) & x_7(1) & x_3(1) \\ x_3(2) & x_5(2) & x_7(2) \\ x_7(3) & x_5(3) & x_3(3) \end{bmatrix}$$

et $L = [3 \to y_3, 5 \to y_5, 7 \to y_7]$

Cet algorithme d'initialisation permet de conserver les attributs triés dans *Attr*. La complexité en espace est $O((F+1) \times n)$ car il faut stocker F listes d'attributs et la table de hachage pour les étiquettes. La complexité en temps est $O(F \times m + n)$ car il faut visiter les F listes d'attributs des m données d'apprentissage dans E et les positions des n données en entrée du nœud dans I_f.

Détermination de la coupure optimale

L'algorithme est très similaire à celui décrit dans la section 9.2 à la différence qu'il n'est plus nécessaire de trier les données suivant chaque attribut lors de la construction de chaque nœud de l'arbre.

```
1: procédure InitialiseStratégieEM(Attr, L, A_f, I_f)
2:     pour chaque position p dans I_f exécute
3:         Crée une entrée dans L : L[p] = 0
4:     fin pour
5:     pour chaque liste d'attributs f_f dans A_f exécute
6:         pour chaque nœud n ∈ f_f exécute
7:             si la position p contenue dans n existe dans L alors
8:                 si c'est la première fois que p est rencontré alors
9:                     L[p] = étiquette contenue dans n.
10:                fin si
11:                Ajoute (x_p(f), p) à la fin de Attr[f].
12:            fin si
13:        fin pour
14:    fin pour
15: fin procédure
```

Algorithme 10.1 – Initialisation de la stratégie EM pour la construction du sous-arbre de décision.

Avant le parcours de chaque liste dans $Attr$, deux histogrammes sont initialisés pour compter la fréquence d'apparition de chaque classe. Le premier, noté H_l, compte les fréquences d'apparition de chaque classe pour les données déjà visitées et le deuxième, noté H_r, pour les données qu'il reste à visiter. Nous notons aussi n_l et n_r le nombre de données visitées et non visitées. Lors du parcours d'une liste d'attributs dans $Attr$, les coupures optimales potentielles sont celles où il y a un changement de valeur suivant l'attribut. La différence d'impureté est alors calculée, basée sur le critère de Gini, comme expliqué dans la section 9.1.1. La somme d'impureté minimum est gardée dans une variable globale notée $\sum I_{op}$. En effet, lorsque pour une nouvelle coupure s suivant un attribut f, nous avons $Gini(E_l) + Gini(E_r) < \sum I_{op}$ alors $\sum I_{op} = Gini(E_l) + Gini(E_r)$ et la règle de décision du nœud devient $x(f) \leq s$. L'algorithme 10.2 décrit comment la coupure optimale est déterminée avec B l'arbre de décision et pos la position du nœud en courant dans l'arbre.

La complexité en espace est $O(n \times (F + 1) + 2 \times K)$ car il faut stocker les listes d'attributs, la table de hachage L et les deux histogrammes contenant K classes. La complexité en temps est $O(F' \times n \times K)$ car pour chaque liste d'attributs, nous parcourons chaque valeur et pour chaque valeur nous pouvons potentiellement calculer $Gini(E_l) + Gini(E_r)$. Nous notons une amélioration par rapport à l'algorithme 9.2 car il n'est plus nécessaire de trier lors de l'exploration d'une nouvelle liste d'attributs.

1: **procédure** DETERMINECOUPUREOPTIMALEEM($Attr$, L, B, pos)
2: $F' \leftarrow$ ensemble d'attributs tirés aléatoirement dans F.
3: $\sum I_{op} \leftarrow$ somme d'impureté minimale.
4: **pour** chaque $f \in F'$ **exécute**
5: $H_l = \emptyset$
6: $H_r =$ fréquence d'apparition des classes dans $Attr$.
7: $n_l = 0$
8: $n_r =$ nombre de données dans $Attr$.
9: $prev \leftarrow$ valeur d'attribut précédente
10: **pour** $i = 1$; $i \leq taille(Attr[f])$; $i = i + 1$ **exécute**
11: $prev = Attr[i-1]$.
12: $(v, p) = (Attr[i], id)$ (valeur et position de la donnée).
13: $H_l[y_p] = H_l[y_p] + 1$ (y_p récupérée avec L).
14: $H_r[y_p] = H_r[y_p] - 1$
15: $n_l = n_l + 1$
16: $n_r = n_r - 1$
17: **si** $v \neq prev$ **alors**
18: $\sum I = Gini(H_l, n_l) + Gini(H_r, n_r)$
19: **si** $\sum I < \sum I_{op}$ **alors**
20: $\sum I_{op} = \sum I$
21: $B[pos].seuil = prev$
22: $B[pos].f_{opt} = f$
23: **fin si**
24: **fin si**
25: **fin pour**
26: **fin pour**
27: **fin procédure**

Algorithme 10.2 – Détermination de la coupure minimale avec la stratégie EM.

Partitionnement des données

Une fois la règle de décision déterminée pour le nœud à la position pos dans l'arbre B, l'étape suivante consiste à distribuer les données aux nœuds fils tout en conservant les listes d'attributs triées. Pour ce faire, nous créons deux nouvelles tables de hachage L_l et L_r contenant les étiquettes des données dans les nœuds fils gauche et droit. Ces tables sont construites en explorant la liste des valeurs de l'attribut f_{opt} sélectionnée pour la règle de décision du nœud à la position pos. Si les données respectent la règle, alors leurs étiquettes sont ajoutées à L_l sinon à L_r. La seconde étape construit les nouvelles listes d'attributs $Attr_l$ et $Attr_r$ en parcourant toutes les listes d'attributs de $Attr$ et en utilisant L_l et L_r. L'algorithme 10.3 décrit l'ensemble de la procédure.

La complexité en espace est $O((2 \times F + 2) \times n)$ car il faut stocker $Attr$, L, $Attr_l$,

1: **procédure** PARTITIONNEMENTEM($Attr$, L, B, pos, $Attr_l$, $Attr_r$, L_l, L_r)
2: **pour** chaque $(v, p) \in Attr\,[B[pos].f_{opt}]$ **exécute**
3: **si** $v \leq B[pos].seuil$ **alors**
4: $L_l[p] = L[p]$
5: **sinon**
6: $L_r[p] = L[p]$
7: **fin si**
8: **fin pour**
9: **pour** chaque $f \in F$ **exécute**
10: **pour** chaque $(v, p) \in Attr[f]$ **exécute**
11: **si** $p \in L_l$ **alors**
12: Ajout de (v, p) à la fin de $Attr_l[f]$.
13: **sinon**
14: Ajout de (v, p) à la fin de $Attr_r[f]$.
15: **fin si**
16: **fin pour**
17: **fin pour**
18: **fin procédure**

Algorithme 10.3 – Partitionnement des données avec la stratégie EM.

$Attr_r$, L_l et L_r en mémoire interne. Cette opération est la plus coûteuse en quantité de mémoire mais elle est nécessaire si nous voulons garder les listes d'attributs triées. La complexité en temps est $O((F + 1) \times n)$ car il faut construire L_l et L_r et ensuite construire $Attr_l$ et $Attr_r$.

Construction récursive de l'arbre de décision avec la stratégie EM

Nous venons de décrire les principales étapes de la construction d'un nœud décisionnel avec la stratégie EM. Nous rappelons que cette stratégie est appelée par la stratégie PEM ou HM lorsque les listes d'attributs et les étiquettes des données peuvent être contenues en mémoire interne. Cette stratégie, exécutée sur un nœud, construit de manière récursive le sous-arbre de décision engendré par ce nœud jusqu'à n'obtenir que des nœuds feuille. L'algorithme 10.4 décrit l'ensemble de la procédure de construction d'un sous-arbre avec la stratégie EM.

Nous supposons que D' est la profondeur du sous-arbre de décision. Nous notons $c = O\left(n \times (F + F' \times K + 1)\right)$ la complexité en temps pour la construction d'un nœud sur un ensemble de données de taille n. L'algorithme est récursif et à chaque niveau l de l'arbre il y a n données à traiter réparties sur 2^l nœuds. Nous supposons que chaque nœud a le même nombre de données en entrée $\frac{n}{2^l}$. Ainsi la complexité en temps pour

1: **procédure** CONSTRUCTIONARBREEM($Attr$, L, B, pos)
2: **si** $B[pos].profondeur == D$ **alors**
3: $B[pos].feuille = VRAI$
4: $B[pos].l =$ classe majoritaire dans $Attr$.
5: Arrête la récursion.
6: **fin si**
7: **si** taille($Attr$)$< Min$ **alors**
8: $B[pos].feuille = VRAI$
9: $B[pos].l =$ classe majoritaire dans $Attr$.
10: Arrête la récursion.
11: **fin si**
12: **si** les données dans $Attr$ sont de la même classe C_k **alors**
13: $B[pos].feuille = VRAI$
14: $B[pos].l = C_k$
15: Arrête la récursion.
16: **fin si**
17: DetermineCoupureOptimaleEM($Attr$, L, B, pos)
18: PartitionnementEM($Attr$, L, B, pos, $Attr_l$, $Attr_r$, L_l, L_r)
19: Ajout du nœud fils gauche $lpos$ dans B.
20: Ajout du nœud fils droit $rpos$ dans B.
21: ConstructionArbreEM($Attr_l$, L_l, B, $lpos$)
22: ConstructionArbreEM($Attr_r$, L_r, B, $rpos$)
23: **fin procédure**

Algorithme 10.4 – Construction récursive de l'arbre de décision avec la stratégie EM.

construire un niveau de l'arbre est :

$$\sum_{i=1}^{2^l} O\left(\frac{n}{2^l} \times (F + F' \times K + 1)\right) = c.$$

Puisque D' est le nombre de niveaux maximum dans l'arbre alors la complexité en temps est $O(D' \times c)$.

10.2.2 Stratégie "Partitionnement En Mémoire"

La stratégie PEM est utilisée lorsque seule la table de hachage P contenant le partitionnement des données dans l'arbre peut être stockée en mémoire interne. P contient des paires clé/valeur où la clé est la position de la donnée dans E et la valeur est la position du nœud dans l'arbre de décision qui contient la donnée. Cette stratégie explore les listes d'attributs triées, notées L_{attr}, dans la mémoire externe pour construire chaque niveau de l'arbre de décision. En effet, comme dans les méthodes SLIQ et SPRINT, l'arbre de

décision est construit de manière transversale pour éviter de multiples lectures dans la mémoire externe. Dans la suite, nous décrivons les différentes étapes de la stratégie PEM en supposant que nous construisons le $t^{\text{ème}}$ arbre de décision B à partir du nœud racine.

Initialisation de P et de l'histogramme initial

Cette étape construit P et la table de hachage contenant les histogrammes initiaux HP qui représentent les fréquences d'apparition des données dans chaque classe pour les nœuds à construire dans l'arbre de décision. Puisque nous commençons avec le nœud racine, seul un nœud est contenu dans HP à la position 0 de l'arbre. L'accès à l'histogramme du nœud racine est noté $HP[0]$ et de manière générale l'accès à un nœud à la position *pos* de l'arbre est noté $HP[pos]$. Pour initialiser HP, seule une liste d'attributs nécessite d'être visitée. Nous rappelons que chaque entrée e_i de la liste d'attributs contient plusieurs informations à savoir l'étiquette y_i, la position i dans E, la valeur $x_i(f)$ suivant l'attribut f et la séquence de bits b_i de longueur T. Lors de la visite de chaque entrée e_i, nous vérifions d'abord si $b_i[t]$ est à 1. Si c'est le cas, alors la donnée x_i est considérée pour la construction de l'arbre de décision. HP est mis à jour en incrémentant le nombre de données appartenant à la classe y_i pour le nœud racine (situé à la position 0 dans B). La paire $(i, 0)$ est ajoutée dans P car nous supposons que toutes les données sont en entrée du nœud racine. L'algorithme 10.5 décrit cette étape d'initialisation.

1: **procédure** INITIALISATIONPEM(HP, P)
2: $HP[0] \leftarrow$ nœud racine dans HP.
3: **pour** chaque $e_i \in L_{attr}[0]$ **exécute**
4: **si** $b_i[t] == 1$ **alors**
5: $HP[0][y_i] = HP[0][y_i] + 1$
6: $P[i] = 0$
7: **fin si**
8: **fin pour**
9: **fin procédure**

Algorithme 10.5 – Initialisation de P et de HP

La complexité en espace est $O(n+K)$ car il faut stocker HP qui contient K classes et P qui contient une paire pour chaque donnée considérée pour la construction de l'arbre. La complexité en temps est $O(m)$ car nous visitons les valeurs suivant un attribut de l'ensemble des données dans E.

Dans tout ce qui suit, nous nous basons sur la construction du niveau courant de l'arbre illustrée dans la figure 10.3.

Sélection aléatoire des attributs

Pour la construction de chaque niveau de l'arbre, F' attributs doivent être sélectionnés aléatoirement parmi F attributs pour chaque nœud à construire. Pour ce faire, une

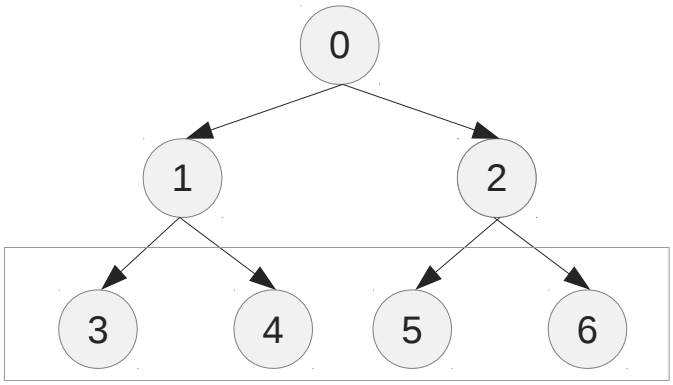


Figure 10.3 – Niveau courant de l'arbre à la profondeur $D = 2$ (identifié par un rectangle rouge) considéré pour illustrer les différentes étapes de la stratégie PEM.

table de hachage notée AP est utilisée. Dans cette table, la clé est l'attribut et la valeur une collection de positions des nœuds pour lesquels l'attribut a été sélectionné. Cette table est construite à partir de HP qui, en considérant le niveau courant de l'arbre dans la figure 10.3, contient 4 clés :

$$HP = \begin{bmatrix} 3 & \rightarrow & H_3 \\ 4 & \rightarrow & H_4 \\ 5 & \rightarrow & H_5 \\ 6 & \rightarrow & H_6 \end{bmatrix} \tag{10.2}$$

Par exemple, si $F = 9$ et $F' = 3$ alors un exemple de tirage aléatoire des attributs pour chaque nœud du niveau courant de l'arbre retourne :

- Pour le nœud 3 : [7, 2, 4].
- Pour le nœud 4 : [1, 8, 5].
- Pour le nœud 5 : [5, 7, 9].
- Pour le nœud 6 : [3, 9, 2].

Par conséquent, AP contient 8 clés :

$$AP = \begin{bmatrix} 1 & \rightarrow & 4 \\ 2 & \rightarrow & 3,6 \\ 3 & \rightarrow & 6 \\ 4 & \rightarrow & 3 \\ 5 & \rightarrow & 4,5 \\ 7 & \rightarrow & 3,5 \\ 8 & \rightarrow & 4 \\ 9 & \rightarrow & 5,6 \end{bmatrix} \tag{10.3}$$

L'algorithme 10.6 décrit la construction de AP. AP a au maximum F entrées contenant des listes de positions de nœuds. Si D est la profondeur du niveau de l'arbre à construire alors HP contient au maximum 2^D nœuds. La complexité en espace est $O(F \times F')$. La complexité en temps est $O(2^D \times F')$ car nous explorons chaque nœud du niveau courant de l'arbre et pour chaque nœud nous explorons chaque attribut sélectionné.

1: **procédure** SÉLECTIONATTRIBUTS(AP, HP)
2: **pour** chaque $(pos, H) \in HP$ **exécute**
3: $F' \leftarrow$ selection de F' attributs parmi F attributs.
4: **pour** chaque $f \in F'$ **exécute**
5: Ajout de pos dans $AP[f]$.
6: **fin pour**
7: **fin pour**
8: **fin procédure**

Algorithme 10.6 – Sélection des attributs avec la stratégie PEM

Détermination des coupures optimales

La détermination des coupures optimales est l'opération la plus complexe dans la stratégie PEM. L'objectif est de déterminer les coupures optimales pour chaque nœud du niveau courant de l'arbre en ne visitant qu'une seule fois les listes d'attributs sélectionnées. En reprenant notre exemple avec la figure 10.3, nous voulons déterminer les coupures optimales des nœuds 3, 4, 5 et 6 en une seule passe sur les listes d'attributs sélectionnées dans AP (définie dans l'équation 10.5). Pour ce faire, 2 tables de hachage des histogrammes HP_l et HP_r sont utilisées où les clés sont les positions des nœuds et les valeurs, leurs histogrammes. HP_l représente les histogrammes pour les données déjà visitées et HP_r pour les données qu'il reste à visiter. HP_l et HP_r sont initialisées pour l'analyse de chaque attribut :

$$HP_l = \begin{bmatrix} 3 & \rightarrow & \emptyset \\ 4 & \rightarrow & \emptyset \\ 5 & \rightarrow & \emptyset \\ 6 & \rightarrow & \emptyset \end{bmatrix} \quad HP_r = \begin{bmatrix} 3 & \rightarrow & HP[3] \\ 4 & \rightarrow & HP[4] \\ 5 & \rightarrow & HP[5] \\ 6 & \rightarrow & HP[6] \end{bmatrix}$$

Pour chaque nœud, des compteurs sont aussi utilisés pour stocker le nombre de données déjà visitées et le nombre de données qu'il reste à visiter. Ces paires de compteurs sont stockées dans deux tables de hachage NP_l et NP_r où la clé est la position du nœud et la valeur le nombre de données. Une table de hachage $Prev$ est utilisée pour stocker la valeur de la donnée précédemment visitée pour chaque nœud. Enfin, une table de hachage I_{opt} est utilisée pour stocker la somme minimale d'impureté pour chaque nœud.

Pour une chaque attribut f dans AP, nous parcourons les entrées dans la liste d'attributs $L_{attr}[f]$. Si l'entrée e_i correspond à une donnée qui doit être considérée pour l'arbre

($b_i[t] = 1$) et si cette donnée appartient à un nœud du niveau de l'arbre $P[i] \in AP[f]$ alors nous mettons à jour les histogrammes $HP_l[P[i]]$ et $HP_r[P[i]]$ du nœud à la position $P[i]$ ainsi que les compteurs $NP_l[P[i]]$ et $NP_r[P[i]]$. Si la valeur de la donnée $x_i(f)$ est différente de celle précédemment visitée $Prev[P[i]]$ alors la somme d'impuretés I est calculée :

$$I = Gini(HP_l[P[i]], NP_l[P[i]]) + Gini(HP_r[P[i]], NP_r[P[i]]) \tag{10.4}$$

Si I est inférieure à la somme d'impureté minimale $I_{opt}[P[i]]$ alors $I_{opt}[P[i]] = I$ et le nœud dans B a pour seuil la valeur précédente $Prev[P[i]]$ et pour attribut optimal l'attribut f. L'algorithme 10.4 résume l'enchaînement de ces différentes étapes pour déterminer les coupures optimales de tous les nœuds du niveau courant de l'arbre.

Beaucoup de tables de hachage sont utilisées pour cet algorithme et chacune d'elles contient au maximum 2^D clés. Cette valeur croît de manière exponentielle avec la valeur de D. À première vue, cet algorithme n'est pas du tout échelonnable. Cependant, il faut garder à l'esprit que la profondeur maximum est donnée par l'utilisateur et dépasse rarement les dizaines, ce qui revient à considérer des milliers de nœuds au maximum. De plus, au fur et à mesure que D augmente, le nombre de nœuds à considérer par niveau diminue et cela pour 2 raisons. La première raison est que certains nœuds aux niveaux antérieurs sont devenus des nœuds feuille et n'ont donc pas de nœuds fils. La seconde raison est que la stratégie d'exécution de certains nœuds aux niveaux antérieurs a changé. Ces nœuds, ainsi que leurs nœuds fils, ne sont donc plus à considérer par la stratégie PEM pour les niveaux suivants. Pour appuyer ces hypothèses, nous avons considéré un ensemble d'apprentissage contenant 3 millions d'échantillons caractérisés par 25 attributs et appartenant à 17 classes distinctes. La description détaillée de ces données d'apprentissage est faite dans l' annexe B. Nous avons considéré seulement la stratégie PEM et nous avons mesuré pour chaque niveau de l'arbre de décision le nombre de nœuds à construire. La figure 10.5 représente le nombre de nœuds suivant la profondeur (niveau) de l'arbre de décision.

Nous pouvons observer, que lors de cette expérience, le nombre de nœuds maximum est atteint à la profondeur égale à 10 et vaut 123. Nous constatons que pour des profondeurs supérieures à 10, le nombre de nœuds par niveau diminue progressivement car de plus en plus de nœuds deviennent des feuilles de l'arbre. De plus, nous ne considérons que la stratégie PEM, ce qui signifie qu'en réalité le nombre de nœuds sera inférieur car certains nœuds pourront être traités avec la stratégie EM. Cette expérience conforte nos hypothèses et nous pouvons supposer que le nombre de nœuds est toujours faible et requiert une quantité de mémoire négligeable.

La complexité en espace est alors $O(2^D \times K + n + F' \times F)$ car il faut stocker les tables de hachage des histogrammes pour chaque nœud, P et AP. La complexité en temps est $O(F \times m \times K)$ car pour chaque attribut dans AP, nous parcourons toutes les entrées dans la liste d'attributs et pour chaque entrée nous pouvons potentiellement calculer la somme d'impureté.

```
procédure DetermineCoupuresOptimales(AP, HP, P, B)
    Prev ← valeurs des données précédemment visitées.
    I_opt ← somme d'impureté minimale.
    HP_l ← histogrammes des données visitées.
    HP_r ← histogrammes des données non visitées.
    NP_l ← nombres de données visitées pour chaque nœud.
    NP_r ← nombres de données non visitées pour chaque nœud.
    pour chaque f ∈ AP exécute
        Initialise HP_l, HP_r, NP_l et NP_r avec HP.
        pour chaque e_i ∈ L_attr[f] exécute
            si b_i[t] == 1 alors
                si P[i] ∈ AP[f] alors
                    HP_l[P[i]][y_i] = HP_l[P[i]][y_i] + 1
                    HP_r[P[i]][y_i] = HP_r[P[i]][y_i] − 1
                    NP_l[P[i]] = NP_l[P[i]] + 1
                    NP_r[P[i]] = NP_r[P[i]] − 1
                    si Prev[P[i]] ≠ x_i(f) alors
                        calcule I(HP_l[P[i]], NP_l[P[i]], HP_r[P[i]], NP_r[P[i]])
                        si I < I_op[P[i]] alors
                            I_op[P[i]] = I
                            B[P[i]].seuil = Prev[P[i]]
                            B[P[i]].f_opt = f
                        fin si
                        Prev[P[i]] = x_i(f)
                    fin si
                fin si
            fin si
        fin pour
    fin pour
fin procédure
```

Figure 10.4 – Détermination des coupures optimales de tous les nœuds d'un même niveau de l'arbre avec la stratégie PEM

Construction de la nouvelle table des histogrammes et mise à jour de l'arbre

Cette opération consiste à établir la nouvelle table de hachage des histogrammes HP avec les nœuds fils des nœuds du niveau courant pour lesquels les coupures optimales ont été déterminées à l'étape précédente. Une nouvelle table de hachage AAP est construite pour stocker, pour chaque attribut f dans AP, la liste des nœuds pour lesquels la coupure

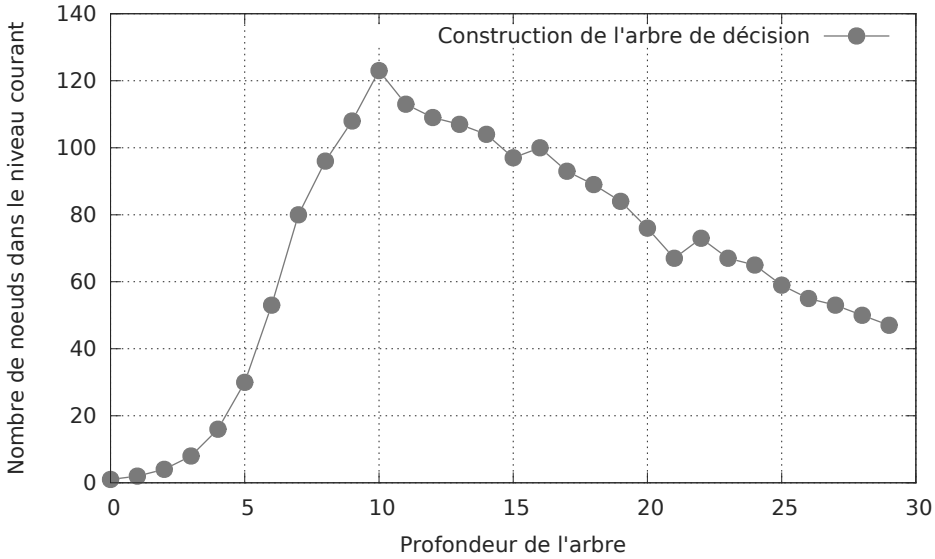


Figure 10.5 – Évolution du nombre de nœuds à construire par niveau en fonction de la profondeur de l'arbre

optimale est effectuée suivant f. AAP est nécessaire pour l'étape suivante qui consiste à distribuer les données aux nouveaux nœuds fils. En reprenant notre exemple de la section 10.2.2, supposons que l'attribut sélectionné pour le nœud 3 est 2, pour le nœud 4 est 8, pour le nœud 5 est 7 et pour le nœud 6 est 2. Alors, AAP s'écrit :

$$AAP = \begin{bmatrix} 2 & \rightarrow & 3,6 \\ 7 & \rightarrow & 5 \\ 8 & \rightarrow & 4 \end{bmatrix} \tag{10.5}$$

Enfin, les nœuds fils sont ajoutés dans l'arbre. L'algorithme 10.7 décrit cette étape.

A l'issue de cette étape, la structure de l'arbre de décision est mise à jour. Les changements sont illustrés par la figure 10.6. HP s'écrit alors :

$$HP = \begin{bmatrix} 7 & \rightarrow & H_7 \\ 8 & \rightarrow & H_8 \\ 9 & \rightarrow & H_9 \\ 10 & \rightarrow & H_{10} \\ 11 & \rightarrow & H_{11} \\ 12 & \rightarrow & H_{12} \\ 13 & \rightarrow & H_{13} \\ 14 & \rightarrow & H_{14} \end{bmatrix} \tag{10.6}$$

La complexité en espace est $O(2^{D+1} \times K + F)$ car il faut stocker les histogrammes des nœuds au niveau dont la profondeur est $D+1$ plus les F attributs potentiels dans AAP.

1: **procédure** MiseAJourPEM(AAP, HP, AP, B)
2: **pour** chaque paire $(f, positions) \in AP$ **exécute**
3: **pour** chaque $pos \in positions$ **exécute**
4: **si** $B[pos].f_{opt} == f$ **alors**
5: Ajout de pos dans $AAP[f]$.
6: Ajout de 2 nœuds fils à $B[pos]$.
7: Ajout des positions des 2 nœuds fils dans HP.
8: **fin si**
9: **fin pour**
10: **fin pour**
11: **fin procédure**

Algorithme 10.7 – Création de AAP, mise à jour de HP et ajout des nœuds fils.

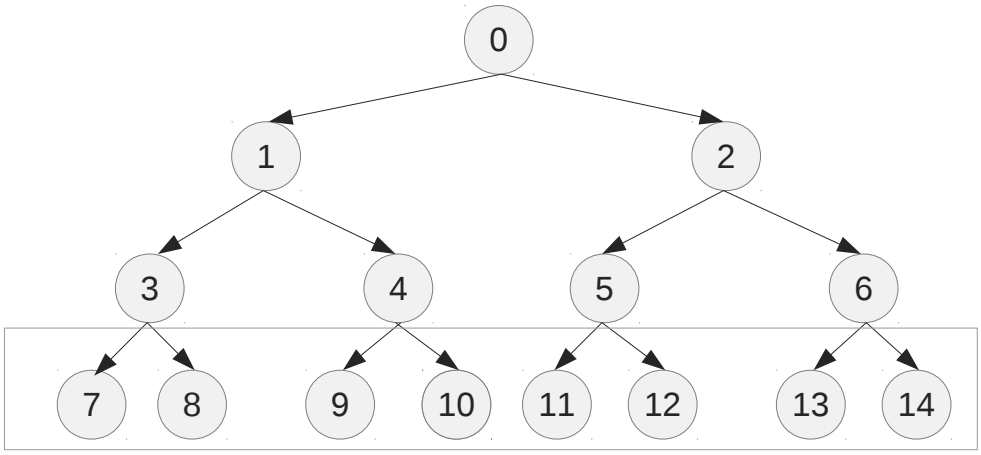


Figure 10.6 – Ajout des nœuds fils aux nœuds du niveau courant

Partitionnement des données aux nœuds fils

Cette étape a pour rôle de mettre à jour P en distribuant les données contenues dans les nœuds du niveau courant aux nœuds fils et de calculer dans le même temps les fréquences d'apparition des classes dans chaque nœud fils. Cette opération s'effectue grâce à AAP qui permet d'explorer seulement les listes dont les attributs ont été sélectionnés lors de la détermination des coupures optimales.

Pour chaque attribut f dans AAP, nous parcourons chaque entrée e_i dans $L_{attr}[f]$. Si $b_i[t] = 1$ alors nous récupérons la position du nœud dans l'arbre pos où la donnée x_i est située. Si pos se trouve dans la collection des positions $AAP[f]$ alors nous analysons si x_i vérifie la règle de décision du nœud $B[pos]$. Si c'est le cas, x_i est distribuée au nœud fils gauche sinon au nœud fils droit. L'algorithme 10.8 décrit l'enchaînement de toutes ces opérations pour effectuer le partitionnement avec la stratégie PEM.

La complexité en espace est $O(2^{D+1} \times K + F \times F' + n)$ car il faut stocker en mémoire

```
1:  procédure PARTITIONNEMENTPEM(P, HP, AAP, B)
2:      pour chaque f ∈ AAP exécute
3:          pour chaque e_i ∈ L_attr[f] exécute
4:              si b_i[t] == 1 alors
5:                  pos = P[i]
6:                  lpos = nœud fils gauche de B[pos].
7:                  rpos = nœud fils droit de B[pos].
8:                  si pos ∈ AAP[f] alors
9:                      si x_i(f) ≤ B[pos].seuil alors
10:                         P[i] = lpos
11:                         HP[lpos][y_i] = HP[lpos][y_i] + 1
12:                     sinon
13:                         P[i] = rpos
14:                         HP[rpos][y_i] = HP[rpos][y_i] + 1
15:                     fin si
16:                 fin si
17:             fin si
18:         fin pour
19:     fin pour
20: fin procédure
```

Algorithme 10.8 – Partitionnement des données avec la stratégie PEM.

interne P, HP et AAP. La complexité en temps est $O(F \times m)$ car pour chaque attribut f dans AAP, il faut parcourir l'ensemble des entrées dans $L_{attr}[f]$.

Détection des nœuds feuille et changement de stratégie

La dernière étape consiste à détecter parmi les nœuds présents dans HP si certains sont des nœuds feuille ou si le nombre de données en entrée permet de changer de stratégie et de passer le relais à la stratégie EM. Un nœud devient une feuille si les données en entrée respectent l'une des 3 conditions que nous rappelons ci-dessous :

- La profondeur maximum est atteinte.
- Toutes les données appartiennent à la même classe
- Le nombre de données est inférieur à Min.

Si l'une de ces conditions est respectée alors le nœud correspondant devient une feuille et prend comme valeur la classe majoritaire des données. La position du nœud est alors supprimée de HP.
Si le nœud n'est pas une feuille, nous vérifions si la taille des données permet de passer le relais à la stratégie EM, c'est-à-dire que l'ensemble des attributs des données et leurs

étiquettes peuvent être stockés en mémoire interne (voir la discusion de la section 10.2.1). Si c'est le cas, alors la position du nœud est stockée dans la collection des nœuds EM et elle est ensuite supprimée de HP. La collection des noeuds EM représente l'ensemble des noeuds qui doivent être exécutés avec la stratégie EM. L'algorithme 10.9 décrit cette étape.

1: **procédure** DETECTIONFEUILLESETNOEUDSEM(HP, B, EM)
2: **pour** chaque $(pos, H) \in HP$ **exécute**
3: **si** pos est un nœud feuille **alors**
4: (pos, H) est supprimée de HP.
5: **fin si**
6: **si** le nœud peut être exécuté avec la stratégie EM **alors**
7: (pos, H) est supprimée de HP.
8: pos est ajoutée dans EM.
9: **fin si**
10: **fin pour**
11: **fin procédure**

Algorithme 10.9 – Détection des nœuds feuille et des nœuds pouvant être exécutés avec la stratégie EM.

La complexité en espace est $O(2^{D+1} \times K)$ car il faut stocker les positions avec les histogrammes de chaque nœud fils au niveau de profondeur $D+1$ de l'arbre de décision. La complexité en temps est $O(2^{D+1})$ car il faut parcourir les positions de chaque nœud fils à la profondeur $D+1$ de l'arbre.

Algorithme de construction d'un arbre de décision avec la stratégie PEM

Nous venons de décrire les différentes étapes engendrées par la construction d'un niveau de l'arbre de décision. L'algorithme de construction d'un arbre de décision répète ces étapes en construisant plusieurs niveaux de l'arbre. La procédure s'arrête lorsque HP ne contient plus de nœuds à construire. Cela signifie que tous les nœuds sont des feuilles de l'arbre ou que les nœuds qui restent à construire peuvent être exécutés avec la stratégie EM. L'algorithme de construction d'un arbre de décision avec la stratégie PEM est décrit dans 10.10.

10.2.3 Stratégie "Hors Mémoire"

Motivation

Nous attirons l'attention du lecteur sur le fait que cette stratégie n'a pas été implémentée et testée car il a été toujours possible de stocker la table de partitionnement P

1: **procédure** ConstructionArbrePEM(B)
2: $HP \leftarrow$ contient les histogrammes des nœuds.
3: $P \leftarrow$ distribution des données dans l'arbre.
4: $AP \leftarrow$ attributs sélectionnés pour déterminer les coupures optimales.
5: $AAP \leftarrow$ attributs sélectionnés par les coupures optimales.
6: $EM \leftarrow$ collection des nœuds pour la stratégie EM.
7: InitialisationPEM(HP, P)
8: **tant que** taille(HP) > 0 **exécute**
9: SélectionAttributs(AP, HP)
10: DetermineCoupuresOptimales(AP, HP, P, B)
11: MiseAJourPEM(AAP, HP, AP, B)
12: PartitionnementPEM(P, HP, AAP, B)
13: DetectionFeuillesEtNoeudsEM(HP, B, EM)
14: **fin tant que**
15: **pour** chaque $pos \in EM$ **exécute**
16: Exécute la stratégie EM sur le nœud $B[pos]$.
17: **fin pour**
18: **fin procédure**

Algorithme 10.10 – Construction d'un arbre de décision avec la stratégie PEM.

en mémoire interne. En effet, nous rappelons que P contient des paires clé/valeur où la clé est la position de la donnée dans E et la valeur la position du nœud dans l'arbre où se situe la donnée. Chaque paire nécessite 12 octets en mémoire interne. Sachant que nous disposons de 12 gigaoctets pour la mémoire interne, nous pouvons stocker exactement 1.073.741.824 paires. Or, jusqu'à présent, nous n'avons pas à gérer des volumes de données d'apprentissage dépassant le milliard de vérités terrain, ce qui a permis à chaque fois de commencer avec la stratégie PEM.

Cependant, dans le futur cette situation peut se présenter, comme par exemple pour la production de cartes d'occupation des sols à échelle globale. C'est pourquoi, nous proposons une stratégie pour manipuler une table de hachage P, qui ne peut être stockée en mémoire interne. Par la suite, nous ne décrivons pas les différentes étapes de construction de l'arbre car elles sont identiques à celles décrites pour la stratégie PEM. Nous nous focalisons seulement sur la structure de données utilisée pour représenter P.

Introduction à la MFU

La principale difficulté réside dans la mise en oeuvre d'une stratégie efficace d'accès à la donnée en mémoire externe. En effet, lors de la détermination des coupures optimales et lors du partitionnement des données (section 10.2.2), l'ordre de visite de ces paires dépend de la liste d'attributs visitée. Puisque chaque liste représente une permutation différente des données, les paires dans P sont consultées aléatoirement. Par conséquent,

cela implique de nombreux accès en lecture et écriture dans la mémoire externe rendant inutilisable l'algorithme sur des volumes de données massifs.

Pour minimiser l'accès à la mémoire externe, nous proposons une structure de données nommée MFU ("Most Frequently Used") pour représenter P. À notre connaissance, cette structure de données n'a jamais été proposée dans la littérature. Nous supposons que chaque donnée a une chance égale d'être consultée lorsque nous parcourons les listes d'attributs. Nous représentons ainsi P avec une table de hachage contenant r clés où chaque clé est un identifiant que nous définirons par la suite. À chaque clé c dans P, nous associons une table de hachage notée P_c contenant q clés. Chaque clé de P_c représente la position d'une donnée dans E et la valeur associée est la position du nœud qui contient la donnée dans l'arbre. La figure 10.7 représente la MFU.

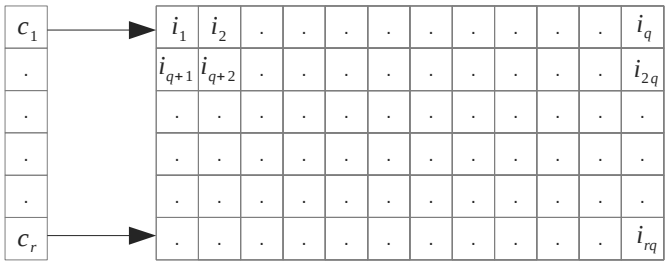


Figure 10.7 – Représentation de la MFU. C'est une table de hachage contenant r clés associées à r tables de hachage P_{c_j}. Chaque table P_{c_j} contient q clés représentant la position des données dans E associées à la position du nœud dans l'arbre où se situe la donnée.

Supposons que nous puissions stocker en mémoire interne N paires représentant le partitionnement des données dans l'arbre et qu'il y ait $M > N$ paires. Avec la définition de la MFU, nous devons utiliser toute la mémoire mise à disposition, c'est-à-dire que $r \times q = N$.

Détermination de la fonction de hachage

La première étape consiste à définir des clés c_j dans la MFU. Nous voulons définir pour chaque table P_{c_j} de taille q une clé c_j unique de telle façon que pour n'importe quel élément dans P_{c_j} nous soyons capables de retrouver la clé c_j. En effet, si e_i est une entrée rencontrée en parcourant une liste d'attributs et que nous n'avons pas sa localisation dans l'arbre (i, pos) en mémoire interne alors nous devons avoir une fonction qui nous retourne une clé permettant de récupérer dans la mémoire externe la table P_{c_j} qui contient (i, pos). Si q est la taille de chaque table P_{c_j} alors il y a au minimum $\lceil \frac{M}{q} \rceil$ tables P_{c_j} à stocker en mémoire externe.
L'idée est d'utiliser une fonction de hachage, qui étant donnée chaque position (i, pos) dans P_{c_j}, retourne c_j. Pour cela, nous proposons la fonction de hachage suivante :

$$H(i, p) : i \equiv c_j \pmod p \tag{10.7}$$

où c_j est le reste de la division de i par p avec p le premier nombre premier supérieur à $\lceil \frac{M}{q} \rceil$. Le fait d'utiliser un nombre premier va permettre d'obtenir p tables P_{c_j} contenant un nombre de paires $q' \leq q$.
Déterminer p s'effectue très facilement en utilisant le Crible d'Ératosthène [107]. Nous avons fait plusieurs simulations pour des valeurs différentes de M avec $q = 25$. Pour chaque valeur de M nous retournons le nombre premier utilisé pour la fonction de hachage ainsi que les nombres minimum et maximum de valeurs pour chaque clé. La table 10.1 présente les nombres de paires minimum et maximum pour chaque table de partitionnement P_{c_j}. Nous voyons que ces nombres sont toujours inférieurs ou égaux à q et les écarts entre le nombre minimum et le nombre maximum de paires sont très faibles (toujours égaux à 1 dans notre simulation), ce qui signifie que les paires sont distribuées uniformément dans chaque table. Ceci s'explique par le fait que nous avons choisi un nombre premier dans la table de hachage. De plus, retrouver la table (c_j, P_{c_j}) pour une paire (i, pos) est immédiat car il suffit de calculer le reste de la division de i par p.

M	$\lceil \frac{M}{q} \rceil$	p	min	max
401	16	17	23	24
500	20	23	21	22
1000	40	41	24	25
7000	280	281	24	25
10000	400	401	24	25

Table 10.1 – Simulation pour différentes valeurs de M en imposant de ne pas avoir plus de $q = 25$ paires dans chaque table de partitionnement P_{c_j}.

Stratégie de remplacement dans la MFU

Pour la stratégie HM, nous proposons la démarche suivante :

- sachant le nombre de données d'apprentissage M pour la construction d'un arbre de décision et sachant que nous pouvons stocker au plus N paires en mémoire interne, la première étape consiste à choisir les valeurs r et q judicieusement telles que $r \times q = N$ avec r le nombre de tables de partitionnement P_{c_j} en mémoire interne et q le nombre de paires dans chaque table ; les r tables de partitionnement P_{c_j} constituent la MFU ;

- la seconde étape consiste à déterminer la fonction de hachage $H(i, p)$ qui génère pour chaque P_{c_j} sa clé c_j ; les p tables de hachage P_{c_j} sont construites à partir de la visite d'une liste d'attributs et sont stockées dans la mémoire externe avec un nom de fichier sous la forme "c_j.bin" avec $c_j \in [0, p-1]$;

Lors de la détermination des coupures optimales où de la mise à jour du partitionnement, nous sommes amenés à visiter les listes d'attributs pour lire la position de la

donnée dans l'arbre ou pour écrire la nouvelle position de la donnée dans l'arbre. Nous rappelons que l'objectif est de minimiser les accès en lecture et écriture dans la mémoire externe. Pour cela, nous associons à chaque table P_{c_j} le nombre de paires déjà consultées f_{c_j}. Lors de la visite de chaque liste d'attributs, chaque paire n'est consultée qu'une seule fois. Par conséquent, si pour P_{c_j} nous avons $f_{c_j} = n_{c_j}$ avec n_{c_j} le nombre de paires dans P_{c_j}, alors nous savons que cette table de hachage ne sera plus consultée par la suite. Dans cette situation, la table est tout simplement supprimée de la MFU. Supposons que la MFU est pleine, c'est-à-dire qu'elle contienne déjà r tables de partitionnement et que nous voulions accéder à la position d'une donnée e_i dans l'arbre qui ne soit pas contenue dans les tables de la MFU. Dans cette situation, il est nécessaire d'enlever une table de la MFU et de l'échanger avec $P_{c_j}(i)$. Une première approche naïve consisterait à maintenir dans la MFU les tables qui ont le plus de chances d'être consultées par la suite. Ainsi, si f_{c_j} est le nombre de paires utilisées dans P_{c_j} alors la probabilité que P_{c_j} soit consultée est :

$$P(P_{c_j}) = \frac{n_{c_j} - f_{c_j}}{M - \sum_{k=1}^{F} n_{c_k}} \tag{10.8}$$

où n_{c_k} est le nombre de paires dans une table où toutes les paires ont déjà été consultées. Si l'apparition des données suit une distribution de probabilité différente de la distribution de probabilité uniforme, alors il suffirait de la prendre en compte dans l'équation 10.8. Nous déterminons la table dans la MFU pour laquelle la valeur de la probabilité dans l'équation 10.8 est minimum et nous échangeons cette table avec $P_{c_j}(i)$.
Cependant, des stratégies de remplacement plus élaborées pourraient être utilisées. Par exemple, si certains attributs sont corrélés alors l'ordre des données triées suivant ces attributs seraient similaires. Il pourrait être intéressant de prendre en compte cette information pour la stratégie de remplacement. Une autre piste serait de s'inspirer des algorithmes de remplacement des lignes de cache [108].

Passage à la stratégie PEM ou à la stratégie EM

Si pour un nœud de l'arbre, nous détectons que la taille des données en entrée permet de passer à la stratégie PEM où à la stratégie EM, alors un fichier contenant les positions de ces données est écrit dans la mémoire externe. La stratégie PEM utilisera ce fichier pour construire la table de partitionnement P et la stratégie EM utilisera ce fichier pour construire la table des listes d'attributs avec la liste des étiquettes.

10.3 Conclusions

L'algorithme SMART est échelonnable et permet de construire un comité d'arbres de décision à partir de volumes de données d'apprentissage de taille arbitraire. Une première contribution est le fait que les listes d'attributs ne sont triées qu'une seule fois au cours de la procédure grâce à la séquence de bits introduite pour chaque donnée x_i de E et permettant de savoir quand celle-ci doit être considérée. Une seconde contribution est

la combinaison de stratégies de construction d'un arbre de décision suivant la taille des données en mémoire. Nous avons proposé 3 stratégies : EM, PEM et HM couvrant toutes les situations possibles :

- La stratégie EM est utilisée lorsque les listes d'attributs et la liste des étiquettes des données peuvent être stockées en mémoire interne.
- La stratégie PEM est utilisée lorsque seulement la table de partitionnement des données peut être stockée en mémoire interne.
- Enfin la stratégie HM est utilisée lorsque ni les listes d'attributs ni la table de partitionnement ne peuvent être contenues en mémoire interne.

L'algorithme choisit la stratégie initiale pour traiter les données en entrée et permet à une stratégie de passer le relais à des stratégies plus efficaces quand cela est possible. Un protocole de communication générique conçu de manière à minimiser l'écriture dans la mémoire externe permet d'effectuer le relais entre les stratégies. Une dernière contribution est l'introduction de la structure de données MFU ("Most Frequently Used") qui permet de manipuler intelligemment une table de partitionnement ne pouvant contenir en mémoire interne. Elle s'inspire de la stratégie utilisée par les mémoires cache qui consistent à rapatrier en plus de la donnée requise d'autres données qui sont suceptibles d'être modifiées. Cela permet de réduire les accès en lecture et écriture dans la mémoire externe.

Dans le chapitre suivant, nous allons valider la stabilité de l'algorithme SMART en montrant qu'il fournit des arbres de décision identiques à ceux obtenus avec l'algorithme de référence dans le cas où les données peuvent être stockées en mémoire. Nous analysons ensuite l'évolution du temps d'exécution de SMART en fonction de la taille de l'ensemble des données d'apprentissage. Enfin, nous montrons la faisabilité de l'algorithme SMART sur un ensemble d'apprentissage recouvrant toute la zone Sud de la France sur plusieurs dates simulant ainsi les données Sentinel-2.

Chapitre 11

Validation de SMART et expérimentations associées

11.1 Validation de la stabilité de l'algorithme SMART

L'objectif de ce chapitre est de comparer les arbres de décision obtenus avec l'algorithme SMART avec ceux obtenus avec l'algorithme de référence. L'algorithme de référence utilisé est celui décrit dans la section 9.1.

Deux arbres de décision sont identiques si les noeuds sont deux à deux identiques. Si ce sont des noeuds décisionnels, cela veut dire qu'ils possèdent la même règle de décision et les mêmes noeuds fils. Si ce sont des noeuds feuille, cela veut dire qu'ils possèdent la même étiquette. Cette section a pour but d'introduire une méthodologie pour comparer les arbres de décision.

11.1.1 Principales difficultés

Pour comparer 2 arbres de décision, nous devons faire face à plusieurs difficultés. La première est due au fait que l'ordre de construction des noeuds diffère suivant l'algorithme utilisé. En effet, avec l'algorithme de référence, l'arbre de décision est construit en profondeur tandis qu'avec l'algorithme SMART, l'arbre est construit de manière transversale.
Les autres difficultés sont dues à l'introduction de l'aléa lors de la construction d'un arbre de décision. L'ensemble des données utilisé pour la construction de l'arbre est tiré aléatoirement. De plus, un ensemble d'attributs est tiré aléatoirement pour la construction de chaque noeud. Il est donc nécessaire de garantir le même ensemble de données pour les 2 arbres et les mêmes ensembles d'attributs pour les noeuds de chaque arbre. Enfin, il se peut que 2 attributs soient candidats pour une coupure optimale. Il faudra faire attention de choisir le même attribut pour chaque noeud.

11.1.2 Méthodologie proposée

Nous notons T_{ref} l'arbre de décision construit avec l'algorithme de référence et T_{smart} l'arbre de décision construit avec l'algorithme SMART.
La problématique liée à l'ordre d'exécution des noeuds est résolue en utilisant une table d'équivalence, notée $NoeudEquiv$ qui permet d'associer la position d'un noeud dans T_{ref} à la position de son noeud équivalent dans T_{smart}.
La problématique liée aux ensembles d'attributs tirés aléatoirement pour chaque noeud est résolue en considérant une table de hachage $AttrEquiv$ qui associe à la position de chaque noeud dans T_{ref} les attributs sélectionnés pour déterminer la coupure optimale. Pour résoudre l'ordre de traitement des attributs lors de la construction d'un noeud, nous introduisons une étape supplémentaire qui consiste à trier les attributs dans l'ordre croissant. De cette manière, les noeuds équivalents dans T_{ref} et T_{smart} explorent les mêmes attributs dans le même ordre.
Une fois T_{ref} et T_{smart} construits, nous pouvons vérifier la stabilité de l'algorithme SMART en nous assurant qu'ils sont identiques. Nous rappelons de nouveau comment nous vérifions qu'un algorithme échelonnable est stable. Nous devons vérifier que le nombre de noeuds dans T_{ref} est égal au nombre de noeuds dans T_{smart} et que chaque noeud dans T_{ref} est également présent dans T_{smart} pour vérifier la bijection. Pour cela, nous utilisons $NoeudEquiv$ et nous comparons les noeuds équivalents. Plusieurs cas peuvent se présenter :

- L'un est un noeud feuille et l'autre un noeud décisionnel : ils ne sont pas identiques et la comparaison s'arrête.

- N_i et N_j sont deux noeuds décisionnels : ils sont identiques si la règle de décision de N_i est identique à la règle de décision de N_j, c'est-à-dire que l'attribut sélectionné pour la coupure optimale est le même et les valeurs seuil sont égales. Si ce n'est pas le cas alors la comparaison s'arrête.

- N_i et N_j sont des noeuds feuille de l'arbre : ils sont identiques si leurs étiquettes sont égales. Si ce n'est pas le cas, la comparaison s'arrête.

La procédure pour vérifier la stabilité de SMART est résumée dans la figure 11.1.

Pour vérifier la stabilité de l'algorithme SMART, nous avons considéré 3 ensembles d'apprentissage (E_1, E_2, E_3) extraits d'une série temporelle composée de 48 images Landsat 8 et 24 images SPOT 4 (Take 5) sur la région Midi-Pyrénées. Un extrait de cette image synthétisée avec ses vérités terrain est illustré par la figure 11.2.

Chaque image étant composée de 7 canaux, chaque donnée d'apprentissage totalise 504 attributs. Le premier ensemble d'apprentissage contient 3221 données, le second 5705 données et le troisième 12727 données. La profondeur maximum de chaque arbre est fixée à $D = 15$. Le nombre de données minimum par noeud est fixé à $Min = 1$. Le nombre d'attributs par noeud vaut la racine carré du nombre total d'attributs soit $F' = 22$. Le protocole suivi pour la vérification de la stabilité de SMART est celui décrit par la figure 11.1.

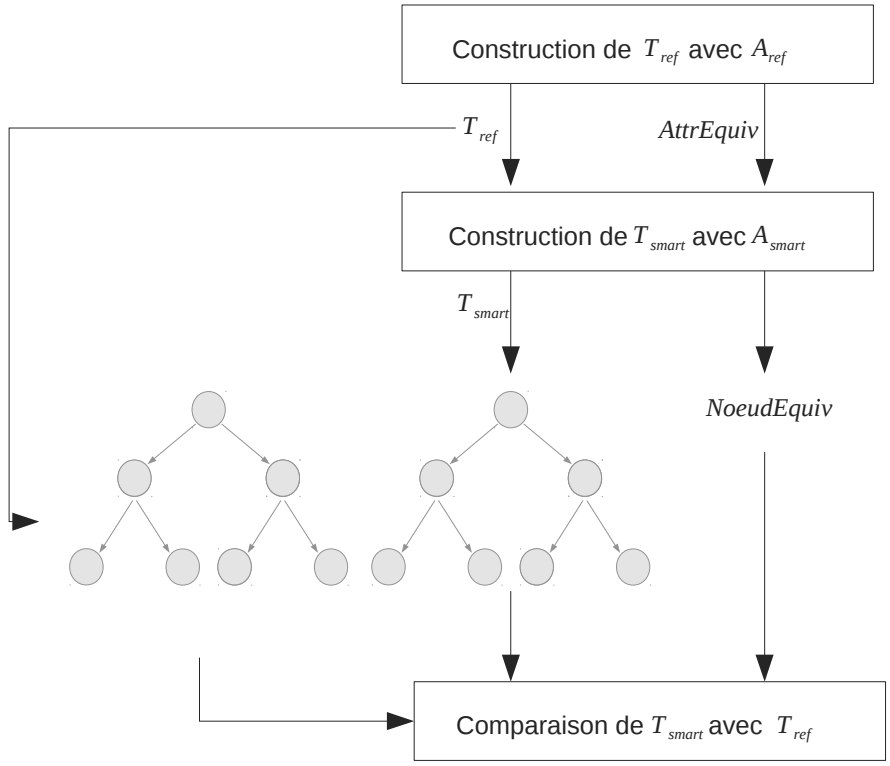
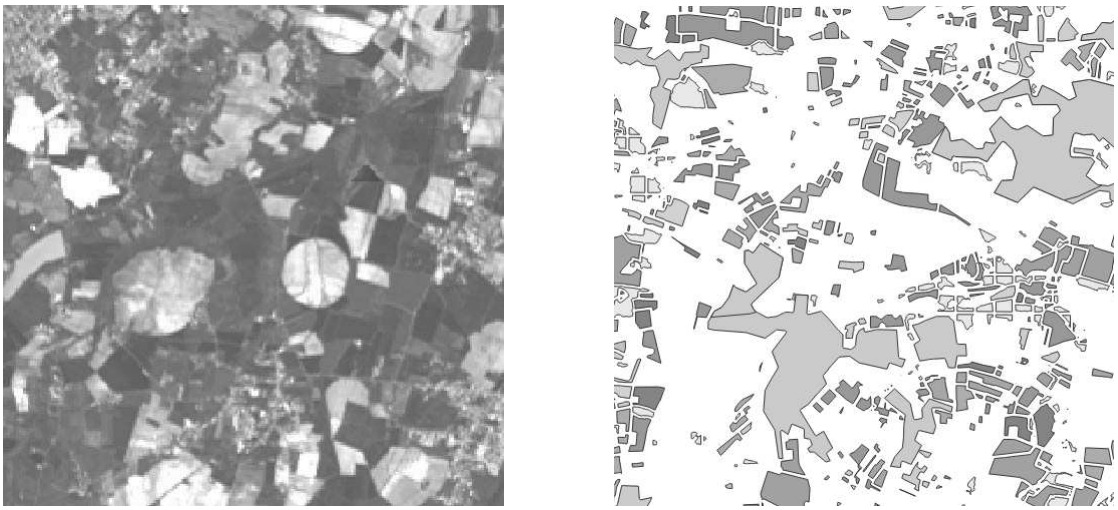


Figure 11.1 – Procédure pour vérifier la stabilité de SMART



(a) un extrait utilisé **(b)** vérités terrain associées

Figure 11.2 – Extrait de taille 1000×1000 pixels issu d'une série temporelle LANDSAT-8 et SPOT 4 (Take 5) contenant 72 dates.

Lors d'un premier apprentissage, seule la stratégie EM est utilisée et fournit l'arbre de décision T_{ref} et la table des attributs $AttrEquiv$. Elle constitue la classification supervisée de référence. Lors du second apprentissage, seule la stratégie PEM est utilisée. Elle nécessite en entrée, en plus des données d'apprentissage, T_{ref} et $AttrEquiv$ pour produire T_{smart} et $NoeudEquiv$. Pour chacun des 3 ensembles d'apprentissage, nous avons toujours obtenu l'égalité $T_{ref} = T_{smart}$, ce qui prouve expérimentalement que les stratégies PEM et HM sont stables.

11.2 Performance de l'algorithme SMART

Dans cette section, nous étudions les performances de l'algorithme SMART sur un ensemble d'apprentissage obtenu à partir de séries multi-temporelles fournies par le satellite Landsat 8 (décrit dans l'annexe B). L'ensemble d'apprentissage contient au total 50 millions de données caractérisées par 150 attributs numériques. Il y a 19 étiquettes distinctes rangeant les données dans des classes thématiques telles que des types de culture, des types de végétation et l'eau. La taille nécessaire pour stocker l'ensemble des données d'apprentissage en mémoire est de 60 gigaoctets.

Pour les tests, nous avons paramétré à 10 Go la quantité de mémoire interne disponible. L'ensemble des tests est effectué avec un seul processeur. Nous avons fixé une profondeur maximale égale à 30 pour chaque arbre de décision. Le nombre d'attributs F' à considérer pour la construction de chaque noeud dans l'arbre de décision est égal à la racine carrée du nombre d'attributs total. Le nombre minimum de données par noeud est fixé à un millième du nombre de données total dans l'ensemble d'apprentissage. Le graphe 11.3 représente l'évolution du temps d'exécution en secondes pour la construction d'un arbre de décision avec l'algorithme SMART en fonction de la taille de l'ensemble des données d'apprentissage. L'évolution du temps d'execution de l'algorithme SMART est linéarithmique, ce qui est logique car la complexité en temps pour la construction d'un arbre de décision est linéarithmique. Nous notons que le temps d'exécution présenté est celui pour la construction d'un seul arbre de décision. Ainsi, pour construire T arbres, il faut multiplier par T ce temps si nous n'utilisons qu'un seul processeur. Cependant, il est intéressant de noter que la construction de chaque arbre peut s'effectuer séparément. De plus, le temps d'exécution pour le tri des données est celui pour la construction de l'ensemble des arbres de décision. En effet, l'algorithme SMART est conçu pour ne trier qu'une seule fois les données.

Le graphe 11.4 représente l'évolution du temps d'exécution en secondes en fonction du nombre d'attributs en fixant le nombre de données d'apprentissage égal à 10 millions, la profondeur maximum de l'arbre de décision à 30 et le nombre minimum de données par noeud à un millième. Si F est le nombre d'attributs alors le nombre d'attributs sélectionné pour chaque noeud est toujours égal à $\sqrt{F}$. Ainsi, l'évolution du temps d'exécution est théoriquement $\sqrt{F} \times n$ où n est le nombre de données d'apprentissage qui est fixé. Sur le graphe, nous observons que la courbe vérifie approximativement cette tendance où par exemple le temps d'exécution de la construction de l'arbre pour $F = 100$ est le

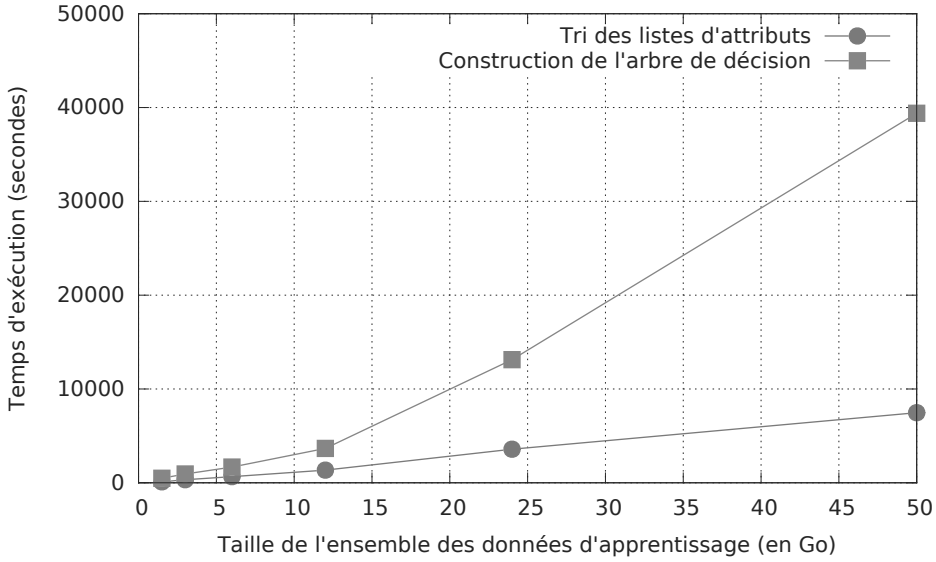


Figure 11.3 – Évolution du temps d'exécution pour la construction d'un arbre de décision avec l'algorithme SMART en fonction de la taille de l'ensemble des données d'apprentissage.

double de celui pour $F = 25$.

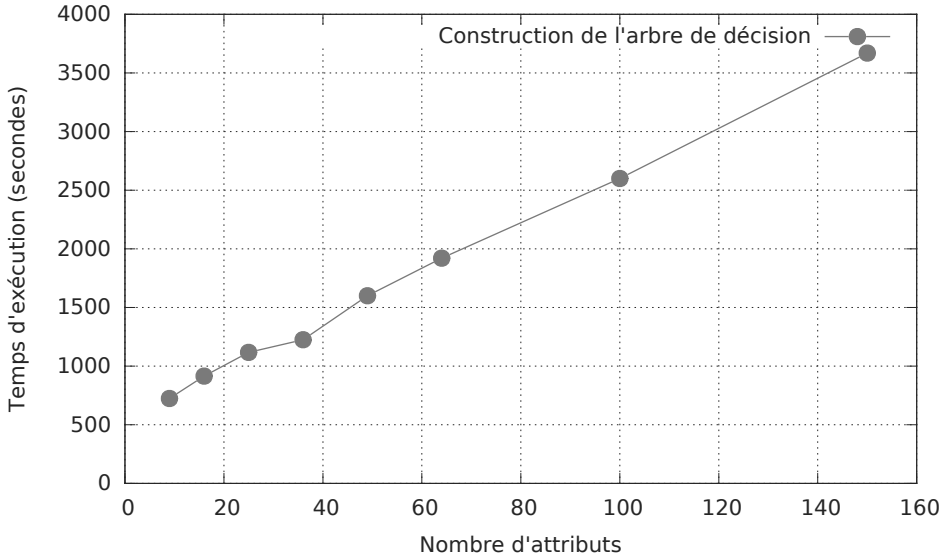


Figure 11.4 – Évolution du temps d'exécution pour la construction d'un arbre de décision avec l'algorithme SMART en fonction du nombre d'attributs pour caractériser chaque donnée d'apprentissage.

Le graphe 11.5 représente l'évolution du temps d'exécution en secondes en fonction de la profondeur maximale de l'arbre de décision. Le nombre de données est toujours égal à 10 millions et le nombre d'attributs est fixé à 25. Ce graphe montre expérimentalement l'apport de l'utilisation de plusieurs stratégies sur le temps d'exécution en fonction de la profondeur de l'arbre. De plus, il permet de comparer notre solution avec les méthodes SLIQ et SPRINT qui n'utilisent que la stratégie PEM. Utiliser seulement la stratégie PEM implique de visiter les listes d'attributs dans la mémoire externe pour la construction de chaque niveau de l'arbre et de calculer pour chaque coupure possible la mesure d'impureté. Lorsque la profondeur maximale de l'arbre est élevée, le nombre de listes d'attributs à visiter croît, ce qui augmente sévèrement le temps d'exécution. C'est la raison pour laquelle les méthodes SLIQ et SPRINT sont peu performantes concernant le temps d'exécution. En revanche, combiner la stratégie PEM avec la stratégie EM permet de réduire significativement le temps d'exécution. Ceci s'explique par le fait que l'accès aux attributs des données stockées en mémoire interne est beaucoup plus rapide.

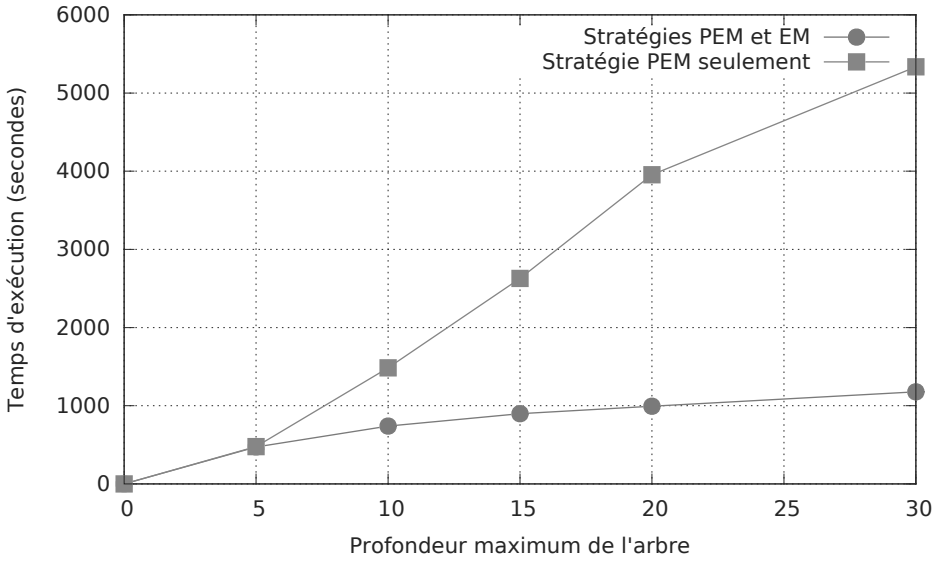


Figure 11.5 – Évolution du temps d'exécution pour la construction d'un arbre de décision avec l'algorithme SMART en utilisant soit seulement la stratégie PEM soit les stratégies PEM et EM en fonction de la profondeur de l'arbre.

Chapitre 12

Bilan sur la classification échelonnable

12.1 Bilan

Cette seconde partie de la thèse s'est attachée à étudier la problématique de l'apprentissage supervisé sur des volumes de données d'apprentissage ne pouvant être stockés dans la mémoire interne tout au long de la procédure. Les solutions existantes dans la littérature proposent, soit de réduire la visibilité de l'algorithme d'apprentissage à un bloc de données pouvant être contenu en mémoire interne, soit d'effectuer de nombreux accès en lecture et écriture dans la mémoire externe. La réduction de la visibilité de l'algorithme induit une baisse de la performance de prédiction du classifieur résultant. Cette solution n'assure pas la stabilité de l'algorithme. La seconde approche présentée par les méthodes SLIQ et SPRINT permet d'assurer un classifieur exact au prix de nombreux accès en lecture et écriture dans la mémoire externe, ce qui augmente drastiquement le temps d'exécution. En effet, pour des volumes de données d'apprentissage de l'ordre des centaines de mégaoctets, la construction d'un seul arbre de décision requiert des dizaines d'heures. Dans notre situation, où nous devons traiter des centaines de gigaoctets et construire plusieurs dizaines d'arbres de décision, ces solutions s'avèrent inutilisables. Nous avons étudié une solution permettant d'assurer la visibilité de l'algorithme des forêts aléatoires sur l'ensemble du volume d'apprentissage tout en minimisant les accès en lecture et écriture dans la mémoire externe.

Le chapitre 9 s'est attaché à étudier le fonctionnement de l'algorithme des forêts aléatoires pour identifier les points critiques concernant la limitation de la mémoire interne. Nous avons par la suite effectué une revue des différentes solutions proposées dans la littérature pour pallier la contrainte de la mémoire.

Après avoir mis en évidence les avantages et les inconvénients des solutions proposées, le chapitre 10 propose une version échelonnable de l'algorithme des forêts aléatoires, baptisée SMART, qui combine plusieurs avantages des solutions proposées dans la littérature tout en essayant de résoudre leurs inconvénients. En particulier, il a été identifié,

dans la version classique de l'algorithme, que le tri des données suivant les attributs est redondant. SMART s'inspire des méthodes SLIQ et SPRINT en permettant à travers le choix judicieux de structures de données de ne trier qu'une seule fois les données suivant chaque attribut. SMART s'inspire aussi de la méthode Google PLANET qui choisit une stratégie d'exécution suivant la taille des données en entrée. La première contribution apportée par SMART par rapport à ces méthodes est le fait de pouvoir construire un comité d'arbres de décision en ne triant qu'une seule fois les données. De cette manière SMART étend les méthodes SLIQ et SPRINT à l'algorithme des forêts aléatoires. Cependant, SMART tente de résoudre le point critique des méthodes SLIQ et SPRINT en minimisant le nombre d'accès en lecture et écriture dans la mémoire externe. En effet, la seconde contribution apportée par SMART est la détection des noeuds de l'arbre de décision qui peuvent être traités en mémoire interne. Pour ces noeuds, SMART change de stratégie d'exécution pour permettre un traitement plus efficace des données.

Finalement, le chapitre 11 a permis de valider la stabilité de l'algorithme SMART avec une métrique proposée qui permet de comparer la structure de 2 arbres de décision pour vérifier qu'ils sont identiques. Il a été montré que l'algorithme SMART fournit des arbres de décision identiques à l'algorithme des forêts aléatoires classique. La faisabilité de l'algorithme SMART a été vérifiée par l'apprentissage supervisé sur des données issues de séries multi-temporelles recouvrant toute la zone sud de la France représentant 60 gigaoctets de données d'apprentissage. L'algorithme SMART ouvre ainsi la perspective de production de cartes d'occupation des sols à l'échelle globale.

12.2 Perspectives

Dans ces travaux, l'objectif était de trouver une solution algorithmique pour pallier la contrainte de la mémoire interne et permettre l'apprentissage supervisé sur des volumes de données d'apprentissage de taille arbitraire.

Par rapport aux méthodes classiques, il a été montré que l'algorithme SMART permet de traiter des volumes de données de l'ordre des dizaines de gigaoctets voire centaines de gigaoctets. Cependant, plusieurs étapes nécessitent d'être améliorées. Le nouveau point critique de l'algorithme SMART se situe lors du relais entre la stratégie PEM et la stratégie EM. En effet, il est nécessaire d'allouer en mémoire les données d'apprentissage qui appartiennent au noeud qu'il faut construire. Cette étape nécessite de lire toutes les listes d'attributs et de sélectionner les valeurs correspondant aux données du noeud. Lorsque le nombre d'attributs est élevé, le temps d'exécution de cette étape devient non négligeable. Une idée serait de charger en mémoire plusieurs ensembles de données pour plusieurs noeuds. Cela impliquerait de retarder le changement de stratégie et de continuer l'exécution de la stratégie PEM jusqu'à que l'ensemble des données des noeuds d'un même niveau puisse être stocké en mémoire interne. Cela permettrait de n'explorer qu'une seule fois toutes les listes d'attributs.
Une seconde idée d'amélioration serait la parallélisation à plusieurs niveaux de l'algorithme SMART. La parallélisation haut niveau consiterait à déployer la construction de

chaque arbre de décision sur plusieurs machines. Ainsi, le temps d'exécution serait divisé exactement par le nombre de machines car aucune communication n'est nécessaire entre les machines durant la procédure de construction. La parallélisation bas niveau concernerait les étapes internes à la construction d'un arbre de décision. En effet, dans l'algorithme SMART, plusieurs étapes lors de la construction d'un arbre de décision sont facilement parallélisables. En particulier, les parallélisations de la détermination des coupures optimales dans la stratégie PEM et de l'exécution des stratégies EM permettraient une réduction non négligeable du temps d'exécution.
Enfin, il serait intéressant de mettre en oeuvre et tester la stratégie HM avec l'utilisation de la structure MFU. En particulier, il faudrait determiner, en fonction de la nature des données d'apprentissage, quelle stratégie de remplacement utiliser pour minimiser les accès dans la mémoire externe.

Conclusions

Chapitre 13

Conclusions générales

Cette thèse s'est focalisée sur une problématique majeure concernant l'exploitation de grands volumes de données en télédétection. En effet, avec l'arrivée des missions à forte revisite temporelle et haute résolution spatiale, un nouveau cap est franchi en terme de volumes de données à exploiter. La problématique se situe au niveau de la limitation de la mémoire disponible dans un ordinateur. Cette contrainte réduit la visibilité de l'algorithme sur les données qu'il doit traiter, modifiant ainsi le résultat final par rapport au résultat attendu. Cette thèse a abordé cette problématique pour les algorithmes de segmentation et de classification en télédétection.

13.1 Segmentation

13.1.1 Bilan

La première partie s'est attachée à étudier le passage à l'échelle des méthodes de segmentation lorsque l'image à segmenter est trop volumineuse pour être stockée dans la mémoire interne de l'ordinateur. Notre attention s'est portée sur les algorithmes de segmentation par fusion de régions car ils sont très utilisés actuellement pour l'analyse orientée objet des images satellites.

Après avoir identifié les étapes communes lors de la procédure d'une segmentation par fusion de régions, nous avons développé un algorithme de fusion de régions générique, baptisé algorithme GRM ("Generic Region Merging"), indépendant du critère local d'homogénéité et de l'heuristique de décision de fusion choisis par l'utilisateur. L'algorithme GRM a été conçu pour utiliser efficacement la ressource mémoire de l'ordinateur avec particulièrement une représentation optimisée des contours des segments.

Nous avons mesuré l'impact du découpage de l'image sur les segments résultants obtenus avec l'algorithme GRM. La présence d'artefacts a été observée sur les bordures communes des tuiles à cause de l'incompatibilité des segments. Une étude des segments à l'intérieur des tuiles a révélé également la présence de segments sous-optimaux. Il a été conclu que le découpage de l'image engendre des segments résultants différents comparés

à ceux obtenus sans un découpage de l'image préalable. L'algorithme GRM n'est donc pas stable lorsqu'une stratégie de découpage est utilisée.

Après avoir identifié les interactions qui peuvent exister entre les données lors d'une procédure de segmentation avec l'algorithme GRM, une version échelonnable, baptisée LSGRM, ("Large Scale Generic Region Merging") a été proposée. Le découpage de la tuile est toujours utilisé. Cependant, la façon de découper les tuiles a été revisitée en considérant autour de chacune d'elles une couronne de stabilité. Cette couronne assure que les segments situés à l'intérieur de la tuile sont identiques à ceux obtenus sans le découpage. La largeur de celle-ci représente la valeur de la marge de stabilité qui est exprimée en fonction du nombre d'itérations que nous désirons appliquer. Ainsi, l'algorithme LSGRM consiste à découper l'image en tuiles avec leur marge de stabilité et à appliquer successivement des segmentations partielles sur les tuiles en considérant à chaque fois une marge de stabilité appropriée.

La stabilité de l'algorithme LSGRM a ensuite été démontrée expérimentalement pour valider l'expression de la marge de stabilité en fonction du nombre d'itérations. Enfin, la faisabilité de l'algorithme LSGRM a été démontrée par la segmentation de plusieurs images complètes Pléiades à très haute résolution nécessitant des dizaines de gigaoctets en mémoire interne. Nous avons noté l'absence d'artefacts sur les bordures communes des tuiles mettant en évidence la compatibilité des segments de part et d'autre de ces bordures et l'absence de segments sous-optimaux à l'intérieur des tuiles montrant que notre solution garantit des résultats stables.

13.1.2 Perspectives

Plusieurs pistes seraient ouvertes pour améliorer l'algorithme LSGRM. La première serait de paralléliser l'exécution de celui-ci sur des environnements distribués. La parallélisation s'effectuerait sur plusieurs niveaux. La parallélisation haut niveau consisterait à déployer la segmentation d'une image sur un cluster où chaque noeud aurait en charge la segmentation d'une ou plusieurs tuiles. Lors de la conception, une attention toute particulière devrait être portée sur le temps de communication entre les noeuds du cluster pour éviter d'augmenter le temps d'exécution. L'avantage est que le temps d'accès dans la mémoire externe apparaît être négligeable devant le temps nécessaire pour la segmentation de chaque tuile. Cette observation est très intéressante car elle indique que le gain obtenu lors de la parallélisation serait potentiellement important. Une parallélisation bas niveau serait également intéressante pour permettre de gagner en accélération sur certaines étapes internes à la segmentation.

Certaines structures de données ont été choisies pour la conception des algorithmes GRM et LSGRM. Il serait intéressant d'effectuer une étude comparative sur le temps d'exécution en fonction des structures de données utilisées. Ceci peut être effectué facilement grâce à la programmation utilisant des templates en C++.

Enfin, nous pensons qu'il est possible d'étendre la méthodologie utilisée pour concevoir l'algorithme LSGRM à d'autres familles d'algorithmes de segmentation. Le travail réalisé dans [63] pour l'algorithme de segmentation Mean-Shift en est un parfait exemple.

13.2 Classification

13.2.1 Bilan

La seconde partie de cette thèse s'est attachée à étudier le passage à l'échelle des méthodes d'apprentissage supervisé en télédétection. Nous avons porté notre attention sur l'algorithme des forêts aléatoires car il est de plus en plus utilisé ces dernières années à cause de sa facilité de paramétrage, sa vitesse d'execution et de la performance des comités de classifieurs faibles.

Il a été constaté que lorsque le volume des données d'apprentissage est trop volumineux pour être stocké dans la mémoire interne, cela a pour conséquence de dégrader la performance de prédiction du classifieur. La raison est que la limitation de la mémoire impose à l'algorithme d'apprentissage d'avoir la visibilité sur une portion non représentative de l'ensemble des données. Plusieurs solutions pour cet algorithme ont été proposées pour pallier la contrainte de la mémoire mais aucune d'elle ne garantit un algorithme échelonnable et stable.

Nous avons proposé une version échelonnable de l'algorithme des forêts aléatoires, baptisé SMART ("Scalable Multi strAtegy Random Trees"). Lorsque le volume d'apprentissage ne peut être stocké en mémoire, cet algorithme assure d'obtenir un classifieur identique à celui obtenu si la contrainte de la mémoire n'existait pas. Pour cela, l'algorithme SMART étend la solution proposée pour les méthodes SLIQ et SPRINT pour la construction d'un comité d'arbres de décision. Les arbres de décision sont construits par niveau en n'explorant qu'une seule fois les listes d'attributs. De plus, l'opération de tri, qui est appelée pour la construction de chaque noeud dans la version classique de l'algorithme, n'est effectuée qu'une seule fois lors de la procédure. Ceci est possible grâce à un choix judicieux des structures de données. Cependant, SMART apporte une amélioration à ces méthodes existantes, permettant de minimiser drastiquement les accès en lecture et écriture dans la mémoire externe. En effet, inspiré par la solution proposée pour Google PLANET, l'algorithme SMART combine plusieurs stratégies de construction de l'arbre de décision. Lorsque la taille des données le permet, SMART passe le relais à une stratégie d'exécution plus efficace pour accélérer le temps d'exécution.

La stabilité de l'algorithme SMART a été démontrée à l'aide d'une nouvelle méthode pour comparer 2 arbres de décision. La faisabilité de l'algorithme a ensuite été testée sur des dizaines de gigaoctets de données d'apprentissage issues de séries multitemporelles recouvrant toute la zone sud de la France. L'algorithme SMART s'est avéré être performant comparé aux solutions proposées dans la littérature.

13.2.2 Perspectives

Plusieurs pistes restent à être approfondies pour améliorer l'efficacité de l'algorithme SMART. Un point critique a été identifié lors du changement entre la stratégie PEM ("Partitionnement En Mémoire") et la stratégie EM ("En Mémoire") lorsque le nombre d'attributs est élevé. L'allocation des données en mémoire requiert un certain temps car

il est nécessaire d'explorer toutes les listes d'attributs initiales dans la mémoire externe. Il serait donc nécessaire d'accélérer cette étape.

Comme pour la segmentation, une seconde piste serait d'étudier la parallélisation de l'algorithme SMART. Une première solution serait que chaque arbre de décision pourrait être construit indépendamment, rendant réalisable la contruction de comités contenant beaucoup de classifieurs sur des environnement distribués. Une parallélisation bas niveau pourrait être étudiée pour les étapes internes de construction d'un arbre de décision.

Enfin, il serait intéressant de mettre en oeuvre et tester la stratégie HM ("Hors Mémoire") avec l'utilisation de la structure de données MFU ("Most Frequently Used"). En particulier il faudrait étudier la stratégie de remplacement utilisée pour la MFU afin de minimiser les accès dans la mémoire externe.

Bibliographie

[1] P. Martimor, O. Arino, M. Berger, R. Biasutti, B. Carnicero, U. Del Bello, V. Fernandez, F. Gascon, P. Silvestrin, F. Spoto, and O. Sy. Sentinel-2 optical high resolution mission for GMES operational services. In *Geoscience and Remote Sensing Symposium, 2007. IGARSS 2007. IEEE International*, pages 2677–2680, July 2007.

[2] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1) :82–85, Jan 1998.

[3] Nihar R. Mahapatra and Balakrishna Venkatrao. The processor-memory bottleneck : Problems and solutions. *Crossroads*, 5(3es), April 1999.

[4] Antonio J. Plaza and Chein-I Chang. *High Performance Computing in Remote Sensing*. Chapman & Hall/CRC, 2007.

[5] S. Baillarin, L. Lebegue, and P. Kubik. Pleiades-hr system qualification : A focus on ground processing and image products performances, a few months before launch. In *Geoscience and Remote Sensing Symposium,2009 IEEE International,IGARSS 2009*, volume 1, pages I–76–I–79, July 2009.

[6] Julien Michel, Manuel Grizonnet, Arnaud Jaen, Sébastien Harasse, Luc Hermitte, Jonathan Guinet, Julien Malik, and Mickaël Savinaud. Open tools and methods for large scale segmentation of very high resolution satellite images. *Proc. OGRS*, pages 179–184, 2012.

[7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[8] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

[9] Guido Rossum. Python reference manual. Technical report, Amsterdam, The Netherlands, The Netherlands, 1995.

[10] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.) : Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[12] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42) :230–265, 1936.

[13] P.J. Plauger, Meng Lee, David Musser, and Alexander A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[14] Aditya Krishna Menon. Large-scale support vector machines : algorithms and theory. *Research Exam, University of California, San Diego*, 2009.

[15] Susanne Albers. Online algorithms : a survey. *Mathematical Programming*, 97(1-2) :3–26, 2003.

[16] S.Z. Iqbal, H. Gull, and J. Ahmed. Incremental sorting algorithm. In *Computer and Electrical Engineering, 2009. ICCEE '09. Second International Conference on*, volume 2, pages 378–381, Dec 2009.

[17] Donald E. Knuth. *The Art of Computer Programming, Volume 3 : (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[18] R. Dementiev, L. Kettner, and P. Sanders. Stxxl : Standard template library for xxl data sets. *Softw. Pract. Exper.*, 38(6) :589–637, May 2008.

[19] Ian Jolliffe. *Principal Component Analysis*. John Wiley and Sons, Ltd, 2005.

[20] Christian Jutten and Jeanny Herault. Blind separation of sources, part 1 : An adaptive algorithm based on neuromimetic architecture. *Signal Process.*, 24(1) :1–10, August 1991.

[21] R. Archibald and G. Fann. Feature selection and classification of hyperspectral images with support vector machines. *Geoscience and Remote Sensing Letters, IEEE*, 4(4) :674–677, Oct 2007.

[22] Nesrine Chehata, Li Guo, and Clément Mallet. Airborne lidar feature selection for urban classification using random forests. In *Proceedings of the ISPRS Workshop : Laserscanning'09*, pages 207–212, 2009.

[23] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2002.

[24] H. D. Cheng, X. H. Jiang, Y. Sun, and Jing Li Wang. Color image segmentation : Advances and prospects. *Pattern Recognition*, 34 :2259–2281, 2001.

[25] Sreenath Rao Vantaram and Eli Saber. Survey of contemporary trends in color image segmentation. *Journal of Electronic Imaging*, 21(4) :040901–1–040901–28, 2012.

[26] S. Lang. Object-based image analysis for remote sensing applications : modeling reality – dealing with complexity. In Thomas Blaschke, Stefan Lang, and GeoffreyJ. Hay, editors, *Object-Based Image Analysis*, Lecture Notes in Geoinformation and Cartography, pages 3–27. Springer Berlin Heidelberg, 2008.

[27] J. Inglada and J. Michel. Qualitative spatial reasoning for high-resolution remote sensing image analysis. *Geoscience and Remote Sensing, IEEE Transactions on*, 47(2) :599–612, Feb 2009.

[28] M.C. Vanegas, I. Bloch, and J. Inglada. Detection of aligned objects for high resolution image understanding. In *Geoscience and Remote Sensing Symposium (IGARSS), 2010 IEEE International*, pages 464–467, July 2010.

[29] M.C. Vanegas, I. Bloch, and J. Inglada. Alignment and parallelism for the description of high-resolution remote sensing images. *Geoscience and Remote Sensing, IEEE Transactions on*, 51(6) :3542–3557, June 2013.

[30] Elisabeth Schöpfer, Stefan Lang, and Josef Strobl. Segmentation and object-based image analysis. In Tarek Rashed and Carsten Jürgens, editors, *Remote Sensing of Urban and Suburban Areas*, volume 10 of *Remote Sensing and Digital Image Processing*, pages 181–192. Springer Netherlands, 2010.

[31] A. Kunte and A. Bhalchandra. Effective region based segmentation technique for high resolution aerial imagery. In *Image and Graphics, 2009. ICIG '09. Fifth International Conference on*, pages 272–275, Sept 2009.

[32] J. Schiewe. Segmentation of high resolution remotely sensed data-concepts, applications and problems. *International Archives of Photogrammetry Remote Sensing And Spatial Information Sciences*, 34(4) :380–385, 2002.

[33] L. Di Stefano and Andrea Bulgarelli. A simple and efficient connected components labeling algorithm. In *Image Analysis and Processing, 1999. Proceedings. International Conference on*, pages 322–327, 1999.

[34] Dorin Comaniciu and Peter Meer. Mean shift : A robust approach toward feature space analysis. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(5) :603–619, May 2002.

[35] R. Gaetano, G. Masi, G. Poggi, L. Verdoliva, and G. Scarpa. Marker-controlled watershed-based segmentation of multiresolution remote sensing images. *Geoscience and Remote Sensing, IEEE Transactions on*, 53(6) :2987–3004, June 2015.

[36] L. Vincent and P. Soille. Watersheds in digital spaces : an efficient algorithm based on immersion simulations. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 13(6) :583–598, Jun 1991.

[37] R. Adams and L. Bischof. Seeded region growing. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 16(6) :641–647, Jun 1994.

[38] Jianping Fan, D.K.Y. Yau, A.K. Elmagarmid, and W.G. Aref. Automatic image segmentation by integrating color-edge extraction and seeded region growing. *Image Processing, IEEE Transactions on*, 10(10) :1454–1466, Oct 2001.

[39] Irene Fondòn, Carmen Serrano, and Bego ona Acha Pinero. Color-texture image segmentation based on multistep region growing. *Optical Engineering*, 45, May 2006.

[40] Ron Ohlander, Keith Price, and D. Raj Reddy. Picture segmentation using a recursive region splitting method. *Computer Graphics and Image Processing*, 8(3) :313 – 333, 1978.

[41] D. Kelkar and S. Gupta. Improved quadtree method for split merge image segmentation. In *Emerging Trends in Engineering and Technology, 2008. ICETET '08. First International Conference on*, pages 44–47, July 2008.

[42] R. Gaetano, G. Scarpa, and G. Poggi. Hierarchical texture-based segmentation of multiresolution remote-sensing images. *Geoscience and Remote Sensing, IEEE Transactions on*, 47(7) :2129–2141, July 2009.

[43] T.H. Hong and Azriel Rosenfeld. Compact region extraction using weighted pixel linking in a pyramid. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-6(2) :222–229, March 1984.

[44] Chen Zhong, Zhao Zhongmin, Yan Dongmei, and Chen Renxi. Multi-scale segmentation of the high resolution remote sensing image. In *Geoscience and Remote Sensing Symposium, 2005. IGARSS '05. Proceedings. 2005 IEEE International*, volume 5, pages 3682–3684, July 2005.

[45] Martin Baatz and Arno Schäpe. Multiresolution segmentation : an optimization approach for high quality multi-scale image segmentation. *Angewandte Geographische Informationsverarbeitung XII*, pages 12–23, 2000.

[46] G. Meinel and M. Neubert. A comparison of segmentation programs for high resolution remote sensing data. In *International Archives of Photogrammetry and Remote Sensing, XXXV*, pages 1097–1105, 2004.

[47] Tsai-Yun Phillips, Azriel Rosenfeld, and Allen C. Sher. O(log n) bimodality analysis. *Pattern Recognition*, 22(6) :741 – 746, 1989.

[48] Leila M. Garcia Fonseca and Fernando Mitsuo Ii. Satellite imagery segmentation : a region growing approach. In *in VIII Brazilian Symposium on Remote Sensing*, pages 677–680, 1996.

[49] GM Espindola, Gilberto Câmara, IA Reis, LS Bins, and AM Monteiro. Parameter selection for region-growing image segmentation algorithms using spatial autocorrelation. *International Journal of Remote Sensing*, 27(14) :3035–3040, 2006.

[50] GAOP Costa, RQ Feitosa, LMG Fonseca, DAB Oliveira, RS Ferreira, and EF Castejon. Knowledge-based interpretation of remote sensing data with the Interimage system : major characteristics and recent developments. *The international archives of the photogrammetry, remote sensing and spatial information sciences ISPRS*, 38 :4, 2010.

[51] A. Darwish, K. Leukert, and W. Reinhardt. Image segmentation for the purpose of object-based classification. In *Geoscience and Remote Sensing Symposium, 2003. IGARSS '03. Proceedings. 2003 IEEE International*, volume 3, pages 2039–2041, July 2003.

[52] David J Crisp, Peter Perry, and Nicholas J Redding. Fast segmentation of large images. In *Proceedings of the 26th Australasian computer science conference-Volume 16*, pages 87–93. Australian Computer Society, Inc., 2003.

[53] R. Nock and F. Nielsen. Statistical region merging. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(11) :1452–1458, Nov 2004.

[54] F. Calderero and F. Marques. Region merging techniques using information theory statistical measures. *Image Processing, IEEE Transactions on*, 19(6) :1567–1586, June 2010.

[55] S.M. LaValle and S.A. Hutchinson. Bayesian region merging probability for parametric image models. In *Computer Vision and Pattern Recognition, 1993. Proceedings CVPR '93., 1993 IEEE Computer Society Conference on*, pages 778–779, Jun 1993.

[56] K. Haris, S.N. Efstratiadis, N. Maglaveras, and A.K. Katsaggelos. Hybrid image segmentation using watersheds and fast region merging. *Image Processing, IEEE Transactions on*, 7(12) :1684–1699, Dec 1998.

[57] Hongzhi Liu, Qiyong Guo, Mantao Xu, and I-Fan Shen. Fast image segmentation using region merging with a k-nearest neighbor graph. In *Cybernetics and Intelligent Systems, 2008 IEEE Conference on*, pages 179–184, Sept 2008.

[58] F. Moscheni, S. Bhattacharjee, and M. Kunt. Spatio-temporal segmentation based on region merging. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(9) :897–915, Sep 1998.

[59] Herbert Freeman. On the encoding of arbitrary geometric configurations. *Electronic Computers, IRE Transactions on*, EC-10(2) :260–268, June 1961.

[60] Vreda Pieterse, Derrick G. Kourie, Loek Cleophas, and Bruce W. Watson. Performance of c++ bit-vector implementations, 2010.

[61] Hui Zhang, Jason E. Fritts, and Sally A. Goldman. Image segmentation evaluation : A survey of unsupervised methods. *Comput. Vis. Image Underst.*, 110(2) :260–280, May 2008.

[62] Adam Hoover, Gillian Jean-Baptiste, Xiaoyi Jiang, Patrick J. Flynn, Horst Bunke, Dmitry B. Goldgof, Kevin Bowyer, David W. Eggert, Andrew Fitzgibbon, and Robert B. Fisher. An experimental comparison of range image segmentation algorithms. *IEEE Trans. Pattern Anal. Mach. Intell.*, 18(7) :673–689, July 1996.

[63] J. Michel, D. Youssefi, and M. Grizonnet. Stable mean-shift algorithm and its application to the segmentation of arbitrarily large remote sensing images. *Geoscience and Remote Sensing, IEEE Transactions on*, 53(2) :952–964, Feb 2015.

[64] Sanghooh Lee. *An unsupervised hierarchical clustering image segmentation and an adaptative image reconstruction system for remote sensing.* PhD thesis, Univ. Texas Austin, 1990.

[65] Sanghoon Lee. Efficient multistage approach for unsupervised image classification. In *Geoscience and Remote Sensing Symposium, 2004. IGARSS '04. Proceedings. 2004 IEEE International*, volume 3, pages 1581–1584 vol.3, Sept 2004.

[66] A Grote and C Heipke. Road extraction for the update of road databases in suburban areas. *International Archives of Photogrammetry and Remote Sensing*, 37(3) :563–568, 2008.

[67] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8) :888–905, August 2000.

[68] Jan Wassenberg, Wolfgang Middelmann, and Peter Sanders. An efficient parallel algorithm for graph-based image segmentation. In *Proceedings of the 13th International Conference on Computer Analysis of Images and Patterns*, CAIP '09, pages 1003–1010, Berlin, Heidelberg, 2009. Springer-Verlag.

[69] PN Happ, Rodrigo S Ferreira, C Bentes, GAOP Costa, and Raul Q Feitosa. Multiresolution segmentation : a parallel approach for high resolution image segmentation in multicore architectures. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 38(4), 2010.

[70] J. Tilton. Split-remerge method for eliminating processing window artifacts in recursive hierarchical segmentation, March 9 2006. US Patent App. 11/251,530.

[71] J.C. Tilton, G. Marchisio, K. Koperski, and M. Datcu. Image information mining utilizing hierarchical segmentation. In *Geoscience and Remote Sensing Symposium, 2002. IGARSS '02. 2002 IEEE International*, volume 2, pages 1029–1031 vol.2, 2002.

[72] ThalesSehn Körting, EmilianoFerreira Castejon, and LeilaMariaGarcia Fonseca. The divide and segment method for parallel image segmentation. In Jacques Blanc-Talon, Andrzej Kasinski, Wilfried Philips, Dan Popescu, and Paul Scheunders, editors, *Advanced Concepts for Intelligent Vision Systems*, volume 8192 of *Lecture Notes in Computer Science*, pages 504–515. Springer International Publishing, 2013.

[73] Romain Goffe. *Pyramides irrégulières descendantes pour la segmentation de grandes images histologiques*. PhD thesis, Université de Poitiers, 2011.

[74] Z. Yang. Tiling and merging framework for segmenting large images, December 27 2011. US Patent 8,086,037.

[75] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *In PLDI*, pages 12–25, 2011.

[76] J.C. Tilton, Y. Tarabalka, P.M. Montesano, and E. Gofman. Best merge region-growing segmentation with integrated nonadjacent region object aggregation. *Geoscience and Remote Sensing, IEEE Transactions on*, 50(11) :4454–4467, Nov 2012.

[77] Yann Dribault, Karem Chokmani, and Monique Bernier. Monitoring seasonal hydrological dynamics of minerotrophic peatlands using multi-date geoeye-1 very high resolution imagery and object-based classification. *Remote Sensing*, 4(7) :1887, 2012.

[78] Xin Huang and Liangpei Zhang. An adaptive mean-shift analysis approach for object extraction and classification from urban hyperspectral imagery. *Geoscience and Remote Sensing, IEEE Transactions on*, 46(12) :4173–4185, Dec 2008.

[79] Miao Li, Shuying Zang, Bing Zhang, Shanshan Li, and Changshan Wu. A review of remote sensing image classification techniques : The role of spatio-contextual information. *European Journal of Remote Sensing*, 47 :389–411, 2014.

[80] Joe H. Ward. Hierarchical grouping to optimize an objective function. *Journal of the American Statistical Association*, 58(301) :236–244, 1963.

[81] N. Alajlan, N. Ammour, Y. Bazi, and H. Hichri. A cluster ensemble method for robust unsupervised classification of vhr remote sensing images. In *Geoscience and Remote Sensing Symposium (IGARSS), 2011 IEEE International*, pages 2896–2899, July 2011.

[82] L. Bruzzone and D.F. Prieto. Unsupervised retraining of a maximum likelihood classifier for the analysis of multitemporal remote sensing images. *Geoscience and Remote Sensing, IEEE Transactions on*, 39(2) :456–460, Feb 2001.

[83] F.S. Marzano, D. Scaranari, M. Montopoli, and G. Vulpiani. Supervised classification and estimation of hydrometeors from c-band dual-polarized radars : A bayesian approach. *Geoscience and Remote Sensing, IEEE Transactions on*, 46(1) :85–98, Jan 2008.

[84] John A. Richards and Xiuping Jia. *Remote Sensing Digital Image Analysis : An Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 3rd edition, 1999.

[85] P.D. Heermann and N. Khazenie. Classification of multispectral remote sensing data using a back-propagation neural network. *Geoscience and Remote Sensing, IEEE Transactions on*, 30(1) :81–88, Jan 1992.

[86] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3) :273–297, September 1995.

[87] John C. Platt. Advances in kernel methods. chapter Fast Training of Support Vector Machines Using Sequential Minimal Optimization, pages 185–208. MIT Press, Cambridge, MA, USA, 1999.

[88] Thorsten Joachims. Advances in kernel methods. chapter Making Large-scale Support Vector Machine Learning Practical, pages 169–184. MIT Press, Cambridge, MA, USA, 1999.

[89] Nello Cristianini and John Shawe-Taylor. *An Introduction to Support Vector Machines : And Other Kernel-based Learning Methods*. Cambridge University Press, New York, NY, USA, 2000.

[90] Erin L. Allwein, Robert E. Schapire, and Yoram Singer. Reducing multiclass to binary : A unifying approach for margin classifiers. *J. Mach. Learn. Res.*, 1 :113–141, September 2001.

[91] J. Ross Quinlan. *C4.5 : Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[92] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

[93] Leo Breiman. Random forests. *Mach. Learn.*, 45(1) :5–32, October 2001.

[94] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2) :123–140, 1996.

[95] Tin Kam Ho. The random subspace method for constructing decision forests. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 20(8) :832–844, Aug 1998.

[96] ThomasG. Dietterich. An experimental comparison of three methods for constructing ensembles of decision trees : Bagging, boosting, and randomization. *Machine Learning*, 40(2) :139–157, 2000.

[97] Jerome H Friedman. Greedy function approximation : a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[98] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. Sliq : A fast scalable classifier for data mining. In *Proceedings of the 5th International Conference on Extending Database Technology : Advances in Database Technology*, EDBT '96, pages 18–32, London, UK, UK, 1996. Springer-Verlag.

[99] John C. Shafer, Rakesh Agrawal, and Manish Mehta. Sprint : A scalable parallel classifier for data mining. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 544–555, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[100] Johannes Gehrke, Raghu Ramakrishnan, and Venkatesh Ganti. Rainforest - a framework for fast decision tree construction of large datasets. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 416–427, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[101] Ruoming Jin and Gagan Agrawal. Communication and memory efficient parallel decision tree construction. In *SDM*, pages 119–129. SIAM, 2003.

[102] Hyontai Sug. A comprehensively sized decision tree generation method for interactive data mining of very large databases. In Xue Li, Shuliang Wang, and ZhaoYang Dong, editors, *Advanced Data Mining and Applications*, volume 3584 of *Lecture Notes in Computer Science*, pages 141–148. Springer Berlin Heidelberg, 2005.

[103] Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. Planet : Massively parallel learning of tree ensembles with mapreduce. *Proc. VLDB Endow.*, 2(2) :1426–1437, August 2009.

[104] Jeffrey Dean and Sanjay Ghemawat. Mapreduce : Simplified data processing on large clusters. *Commun. ACM*, 51(1) :107–113, January 2008.

[105] Gurmeet Singh Manku, Sridhar Rajagopalan, and Bruce G Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *ACM SIGMOD Record*, volume 28, pages 251–262. ACM, 1999.

[106] Jiawei Han, Yanheng Liu, and Xin Sun. A scalable random forest algorithm based on mapreduce. In *Software Engineering and Service Science (ICSESS), 2013 4th IEEE International Conference on*, pages 849–852, May 2013.

[107] Samuel Horsley. Koσ kinon epatoσ θ enoy σ. or, the sieve of eratosthenes. being an account of his method of finding all the prime numbers, by the rev. samuel horsley, frs. *Philosophical Transactions (1683-1775)*, pages 327–347, 1772.

[108] Hussein Al-Zoubi, Aleksandar Milenkovic, and Milena Milenkovic. Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite. In *Proceedings of the 42Nd Annual Southeast Regional Conference*, ACM-SE 42, pages 267–272, New York, NY, USA, 2004. ACM.

[109] P. Lassalle, J. Inglada, J. Michel, M. Grizonnet, and J. Malik. Large scale region-merging segmentation using the local mutual best fitting concept. In *Geoscience and Remote Sensing Symposium (IGARSS), 2014 IEEE International*, pages 4887–4890, July 2014.

[110] P. Lassalle, J. Inglada, J. Michel, M. Grizonnet, and J. Malik. A scalable tile-based framework for region-merging segmentation. *Geoscience and Remote Sensing, IEEE Transactions on*, 53(10) :5473–5485, Oct 2015.

Annexes

A Description des satellites Pléiades

La table 1 décrit les satellites Pléiades. Les travaux sur la segmentation ont été testées sur les images à très haute résolution fournies par ces satellites.

Nom satellite	Pléiades-1A et Pléiades-1B
Famille	Pléiades
Nationalité	France
Début mission	2011
statut	actif
Masse	980 kg
Bandes (avec résolution)	0.43 μm - 0.55 μm 2.8 m 0.49 μm - 0.61 μm 2.8 m 0.6 μm - 0.72 μm 2.8 m 0.75 μm - 0.95 μm 2.8 m
Pan Résolution	0.48 - 0.83 μm 0.7 m
Altitude Cycle Emprise (km $\times$ km)	694 km 26 jours 20 $\times$ 20

Table 1 – Description des satellites Pléiades.

B Description des données d'apprentissage

Les données d'apprentissage utilisées pour tester la stabilité et la faisabilité de l'algorithme SMART sont issues d'un ensemble de séries multi-temporelles recouvrant toute la zone sud de la France. Au total, nous disposons de 23 tuiles de taille 7189×6997 illustrées dans la figure 1.

Figure 1 – Ensemble des données d'apprentissage recouvrant la zone sud de la France.

Chaque tuile est composée de 15 images multispectrales prises à différentes dates durant l'année 2013 par le satellite Landsat 8 décrit dans la table 2.

Chaque image multispectrale est composée de 7 bandes dont 4 dans le spectre visible (400 nm à 800 nm) et 3 dans le domaine infrarouge. Chaque bande spectrale est fournie avec une résolution de 30 mètres. En plus de de ces bandes spectrales, 3 indices de végétation sont également fournis : le NDVI, NDWI ainsi que l'indice de luminance du sol (*Brightness*). Le NDVI est un indice de végétation par différence normalisée qui est construit à partir de la bande rouge, noté R et de la bande infrarouge, noté PIR et se calcule de la façon suivante :

$$NDVI = \frac{PIR - R}{PIR + R}. \tag{1}$$

Le NDWI est l'indice de teneur en eau par différence normalisée. Il est construit à partir d'une bande infrarouge à courte longueur d'onde, notée $SWIR$ (entre 1500 et 1750 nm) et de la bande proche infrarouge, notée NIR. Il se calcule de la façon suivante :

$$NDWI = \frac{NIR - SWIR}{NIR + SWIR}. \tag{2}$$

Nom satellite	Landsat 8
Famille	Landsat
Nationalité	USA
Début mission	2013
statut	actif
Masse	1512 kg
Instruments	ETM+ et OLI/TIRS
Bandes (avec résolution)	0.450 μm - 0.515 μm 30 m 0.525 μm - 0.605 μm 30 m 0.630 μm - 0.690 μm 30 m 0.775 μm - 0.900 μm 30 m 1.550 μm - 1.750 μm 30 m 2.090 μm - 2.350 μm 30 m 10.00 μm - 12.50 μm 60 m
Pan Résolution	0.520 - 0.900 μm 15 m
Altitude Cycle Emprise (km $\times$ km)	705 km 16 jours 185 $\times$ 180

Table 2 – Description du satellite Landsat 8.

L'indice de luminance du sol est calculé à partir des 7 bandes spectrales disponibles avec le satellite LANDSAT-8 :

$$Brightness = \sqrt{\sum_{i=1}^{7} b_i^2}; \tag{3}$$

où b_i est la $i^{\text{ème}}$ bande spectrale. Au total, chaque image est ainsi composée de 10 canaux.

Une base de données vecteur contenant les vérités terrain de la zone recouverte par les tuiles est utilisée. Au total, cette base de données vecteur contient un peu plus de 3 millions de polygones regroupant 19 classes. Chacun des polygones contient plusieurs pixels. Dans nos tests, une donnée d'apprentissage représente un pixel contenu dans un des polygones de la base de données vecteur. Au total, 47.663.099 pixels composent notre ensemble d'apprentissage.

C Valorisation scientifique

La détermination de la zone d'influence et l'expression de la marge de stabilité pour les méthodes de segmentation par fusion de régions a été valorisée par une présentation orale dans une conférence [109].

Ce même travail a aussi été primé lors des journées CNES Jeunes Chercheurs et a fait l'objet d'un article dans le journal CNES Magazine paru en Janvier 2015 page 31 mettant en avant l'apport de cette recherche dans le domaine du *big data*.

Enfin, l'ensemble des travaux sur la segmentation ont conduit à un article [110] décrivant les différentes étapes de l'algorithme LSGRM.

Un article sur l'algorithme SMART est actuellement en préparation.

D Applications et développement logiciel

L'algorithme GRM est disponible en tant que Remote Module dans le logiciel Orfeo Toolbox [1]. L'algorithme LSGRM ainsi que l'algorithme SMART sont en cours de préparation pour être intégrés en tant que Remote Module dans OTB. Les développements en version beta sont actuellement disponibles sur le site de l'auteur [2].

Ces algorithmes sont actuellement utilisés dans le domaine de la recherche (SERTIT à Strasbourg, l'université Harban Instiutute of Technology en Chine, le projet Gnorasi en Grèce) mais aussi dans le domaine industriel avec l'entreprise C-S Systèmes d'Information et l'entreprise Atos qui porte les algorithmes dans un environnement distribué utilisant la technologie Apache Spark pour le traitement des images Sentinel-2.

1. Orfeo Toolbox est un logiciel libre de traitement d'images satellites développé par le CNES.
2. http://pierre33.github.io/libraries.html

Résumé

Les récentes missions spatiales d'observation de la Terre fourniront des images optiques à très hautes résolutions spatiale, spectrale et temporelle générant des volumes de données massifs. L'objectif de cette thèse est d'apporter de nouvelles solutions pour le traitement efficace de grands volumes de données ne pouvant être contenus en mémoire. Il s'agit de lever les verrous scientifiques en développant des algorithmes efficaces qui garantissent des résultats identiques à ceux obtenus dans le cas où la mémoire ne serait pas une contrainte.

La première partie de la thèse se consacre à l'adaptation des méthodes de segmentation pour le traitement d'images volumineuses. Une solution naïve consiste à découper l'image en tuiles et à appliquer la segmentation sur chaque tuile séparément. Le résultat final est reconstitué en regroupant les tuiles segmentées. Cette stratégie est sous-optimale car elle entraîne des modifications par rapport au résultat obtenu lors de la segmentation de l'image sans découpage. Une étude des méthodes de segmentation par fusion de régions a conduit au développement d'une solution permettant la segmentation d'images de taille arbitraire tout en garantissant un résultat identique à celui obtenu avec la méthode initiale sans la contrainte de la mémoire. La faisabilité de la solution a été vérifiée avec la segmentation de plusieurs scènes Pléiades à très haute résolution avec des tailles en mémoire de l'ordre de quelques gigaoctets.

La seconde partie de la thèse se consacre à l'étude de l'apprentissage supervisé lorsque les données ne peuvent être contenues en mémoire. Dans le cadre de cette thèse, nous nous focalisons sur l'algorithme des forêts aléatoires qui consiste à établir un comité d'arbres de décision. Plusieurs solutions ont été proposées dans la littérature pour adapter cet algorithme lorsque les données d'apprentissage ne peuvent être stockées en mémoire. Cependant, ces solutions restent soit approximatives, car la contrainte de la mémoire réduit à chaque fois la visibilité de l'algorithme à une portion des données d'apprentissage, soit peu efficaces, car elles nécessitent de nombreux accès en lecture et écriture sur le disque dur. Pour pallier ces problèmes, nous proposons une solution exacte et efficace garantissant une visibilité de l'algorithme sur l'ensemble des données d'apprentissage. L'exactitude des résultats est vérifiée et la solution est testée avec succès sur de grands volumes de données d'apprentissage.

Abstract

Recent Earth observation spatial missions will provide very high spectral, spatial and temporal resolution optical images, which represents a huge amount of data. The objective of this research is to propose innovative algorithms to process efficiently such massive datasets on resource-constrained devices. Developing new efficient algorithms which ensure identical results to those obtained without the memory limitation represents a challenging task.

The first part of this thesis focuses on the adaptation of segmentation algorithms when the input satellite image can not be stored in the main memory. A naive solution consists of dividing the input image into tiles and segment each tile independently. The final result is built by grouping the segmented tiles together. Applying this strategy turns out to be suboptimal since it modifies the resulting segments compared to those obtained from the segmentation without tiling. A deep study of region-merging segmentation algorithms allows us to develop a tile-based scalable solution to segment images of arbitrary size while ensuring identical results to those obtained without tiling. The feasibility of the solution is shown by segmenting different very high resolution Pléiades images requiring gigabytes to be stored in the memory.

The second part of the thesis focuses on supervised learning methods when the training dataset can not be stored in the memory. In the frame of the thesis, we decide to study the Random Forest algorithm which consists of building an ensemble of decision trees. Several solutions have been proposed to adapt this algorithm for processing massive training datasets, but they remain either approximative because of the limitation of memory imposes a reduced visibility of the algorithm on a small portion of the training datasets or inefficient because they need a lot of read and write access on the hard disk. To solve those issues, we propose an exact solution ensuring the visibility of the algorithm on the whole training dataset while minimizing read and write access on the hard disk. The running time is analysed by varying the dimension of the training dataset and shows that our proposed solution is very competitive with other existing solutions and can be used to process hundreds of gigabytes of data.