IUT Nancy Charlemagne
Département Informatique
Université de Lorraine
2 ter Boulevard Charlemagne
BP 55227
54052 Nancy Cedex

TRADUCTION D'UN ALGORITHME DE GESTION DE MTREE VERS RUST

Rapport de stage de DUT Informatique
CNRS – Observatoire Astronomique de Strasbourg

Stagiaire : Jérémy RISACHER DAMEN
Tuteur de l’observatoire : François-Xavier PINEAU
Tuteur de l’IUT : Denis ROEGEL
Année Universitaire 2018-2019

TRADUCTION D'UN ALGORITHME DE GESTION DE M-TREE EN RUST

Rapport de Stage de Jérémy RISACHER DAMEN

Remerciements

Je remercie tout d'abord monsieur PINEAU, mon maître de stage de l'observatoire astronomique de Strasbourg, pour avoir créé ce sujet et m'avoir guidé durant les 10 semaines de stage, ainsi que monsieur ROEGEL, mon parrain de stage, pour m'avoir donné des instructions et des conseils pour le rapport de stage et la soutenance.

Je tiens également à remercier monsieur SCHAAFF de l'observatoire pour m'avoir permis d'accéder au stage et madame BELLALEM de l'IUT Nancy Charlemagne pour m'avoir mis en relation avec monsieur SCHAAFF.

Ensuite, je remercie monsieur DUC, directeur de l’observatoire, le secrétariat de l'IUT ainsi que toutes les personnes mentionnées précédemment pour m'avoir accordé des jours dans le cadre de mes recherches de poursuite d'études.

Enfin, je remercie mes camarades stagiaires et le reste de l’observatoire, qui m’ont aidé avec mon sujet et pour ma soutenance.

Sommaire

1 - Introduction .... - 5 -
2 - L'Observatoire Astronomique de Strasbourg .... - 6 -
3 - La problématique du stage .... - 7 -
3.1 - Le contexte et la mission .... - 7 -
3.2 - La structure de données M-Tree .... - 7 -
3.2.1 - L'architecture du MTree .... - 8 -
3.2.2 - Les algorithmes de recherche du MTree .... - 9 -
3.3 - Les langages utilisés, point comparaison .... - 9 -
3.3.1 - Orientation .... - 9 -
3.3.2 – Syntaxe .... - 10 -
3.3.3 - Niveau de langage .... - 11 -
3.3.4 - Gestion de la mémoire .... - 11 -
3.3.5 - Compilation .... - 12 -
3.3.6 - Autres différences .... - 13 -
3.3.7 - Résumé .... - 14 -
3.4 - Coordonnées et distances .... - 14 -
3.4.1 - Coordonnées 3D .... - 14 -
3.4.2 - Coordonnées équatoriales .... - 15 -
3.4.3 - Chaines de caractères .... - 16 -
3.5 - Le M-Tree dans X-Match .... - 17 -
4 - Le déroulement du stage .... - 19 -
4.1 - Les conditions de travail .... - 19 -
4.1.1 - Conditions globales .... - 19 -
4.1.2 - Outils de travail .... - 19 -
4.2 - L'avancement du projet étape par étape .... - 20 -
4.2.1 - Formation à Rust (continue) .... - 22 -

4.2.2 - Étude du programme Java et du MTree (continue) .... - 22 -

4.2.3 - Traduction "brute" (10/04 - 15/04) .... - 22 -

4.3.3 - Écriture d'un MTree de base (16/04 - 07/05) .... - 23 -

4.3.4 - Recherches du MTree de base (09/05 - 17/05) .... - 26 -

4.3.5 – Tests de performances et de traces Java-Rust (22/05 - 28/05) .... - 26 -

4.3.6 - Documentation (29/05 - 31/05) .... - 29 -

4.3.7 – La fin du stage (03/06 – 14/06) .... - 30 -

5 - La conclusion .... - 32 -

6 - Bibliographie & Webographie .... - 33 -

6.1 Bibliographie .... - 33 -

6.2 Webographie .... - 33 -

7 – Annexes .... - 34 -

1 - Introduction

L'Observatoire Astronomique de Strasbourg possède un service X-Match (prononcé cross-match), permettant de croiser des bases de données astronomiques afin de comparer les informations qu'elles contiennent. Un des éléments utilisés par ce service est la structure de données M-Tree permettant de stocker sous forme d'arbre des données spatiales indexées selon leur position et leur distance entre elles.

Le programme permettant de gérer le M-Tree actuellement utilisé par le service X-Match est programmé en langage Java. Pour améliorer les performances de cet algorithme, mon maitre de stage, monsieur PINEAU, m'a confié pour sujet la conversion de son algorithme pour passer d'une rédaction de Java à un langage offrant plusieurs avantages, dont de meilleures performances : Rust.

2 - L'Observatoire Astronomique de Strasbourg

L'Observatoire Astronomique de Strasbourg (abrégé ObAS) est un centre de recherches et de formation en astronomie. Il est une coopération entre le CNRS (Centre National de la Recherche Scientifique) et l'Université de Strasbourg. Deux équipes se distinguent à l'ObAS.

L'équipe GALHECOS (Galaxies, High Energy, Cosmology, Compact Objects & Stars) effectue des études et des recherches sur l'évolution des galaxies, les phénomènes à hautes énergie et la cosmologie.

L'équipe du Centre de Données astronomiques de Strasbourg (abrégé CDS) apporte des outils informatiques pour la recherche astronomique, incluant la mise à disposition de bases de données astronomiques et des services comme VizieR, Simbad, Aladin ou le X-Match.

3 - La problématique du stage

3.1 - Le contexte et la mission

L'ObAS possède un algorithme utilisable en ligne appelé X-Match programmé par mon maitre de stage. Ce service permet de croiser des tables de données astronomiques à partir de positions. En d'autres termes, de combiner de grandes listes contenant des informations sur un nombre volumineux d'astres. Le service de X-Match est mis en production sur un des serveurs du CDS.

L'ObAS possède plusieurs bases de données avec des enregistrements et des informations différentes (exemple : astres observés dans tel ou tel domaine de longueur d'ondes). Permettre de croiser ces bases c'est pouvoir réunir des informations sur de mêmes astres ou vérifier des informations comme les coordonnées d'un corps céleste.

Pour gérer la jointure floue des bases volumineuses de façon rapide et efficace, plusieurs méthodes sont utilisées. L'une d'entre elles est l'usage d'une structure de données appelée le M-Tree. Son fonctionnement est détaillé plus loin dans le rapport.

Le développement de la gestion d'un M-Tree fut initialement réalisé en langage Java. Mais après des études effectuées antérieurement à mon stage, reproduire cette structure en langage Rust semblait être un bon moyen d'améliorer les performances du logiciel et de rendre le code compilable en WebAssembly (le rendant utilisable facilement sur le web) et utilisable en Python ou PSQL. **Ma mission dans le cadre de mon stage était de convertir les algorithmes de gestion de M-Tree réalisés en Java dans le langage Rust.**

3.2 - La structure de données M-Tree

Les informations que j'ai sur le M-Tree me viennent de la référence présente dans la Bibliographie et du lien 4) de la Webographie.

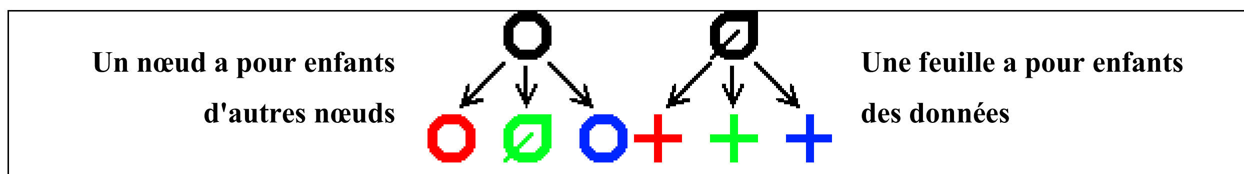
Le M-Tree est une extension du B-Tree, une structure permettant d'indexer des données sur une dimension triées selon un ordre donné. Le M-Tree permet de stocker des données munies de coordonnées spatiales.

Son avantage par rapport à d'autres structures de données spatiales comme le R-Tree est la capacité de stocker non seulement des points, mais aussi des points munis d'une extension spatiale (formant des sphères), ce qui permet de gérer les approximations de coordonnées et la marge d'erreur de mesures.

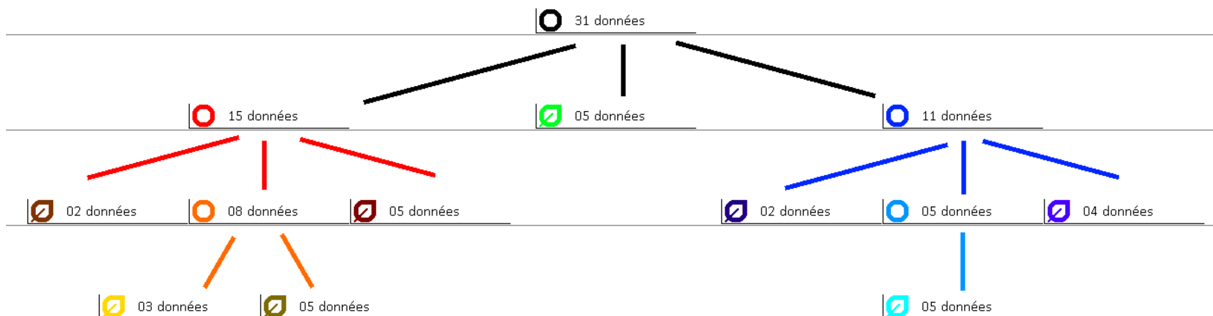
3.2.1 - L'architecture du MTree

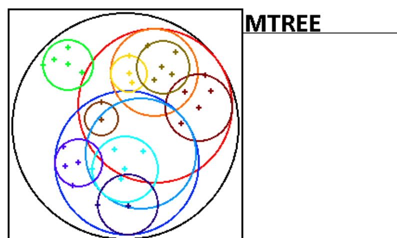
Un MTree est un arbre permettant le stockage de données à partir d'une fonction de calcul de distances. Il se compose de nœuds, et de données. Certains nœuds sont appelés des feuilles.

Chaque nœud de l'arbre peut posséder un à plusieurs "fils" ou "enfants" dont il est le "parent". Les nœuds classiques ont pour enfants d'autres nœuds. Les nœuds dits "feuilles" contiennent des données qui peuvent représenter des points ou des sphères.



Les nœuds possèdent des coordonnées indiquant leur centre et un rayon. Tout enfant est intégralement contenu dans la sphère que représente son parent, qu'il soit un point ou une sphère.





3.2.1.a : Exemple de représentation d'un M-Tree de données en 2 dimensions sous forme d'arbre et de représentation spatiale

3.2.2 - Les algorithmes de recherche du MTree

Pour exploiter le MTree, il faut pouvoir récupérer des informations dans ce dernier. L'avantage du MTree est que des données situées dans une même région sont proches dans l'arbre. On réduit donc grandement la taille de la structure à explorer.

Parmi les algorithmes de recherche dans le MTree qui existent, on peut mentionner :

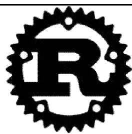
- La **recherche NN** (Neirest Neighbour) qui, traduite, signifie que l'on recherche le plus proche voisin des coordonnées indiquées, soit la donnée la plus proche.
- La **recherche KNN** (K Neirest Neighbour) où, pour un entier positif K et des coordonnées données, on va chercher et ranger dans l'ordre du plus proche au plus éloigné des coordonnées les K éléments les plus proches de ces coordonnées.
- La **range-query** où, pour une sphère indiquée, on recherche tous les éléments contenus dans cette sphère.

Se sont les principaux algorithmes de recherche.

3.3 - Les langages utilisés, point comparaison

Dans le cadre de mon stage, deux langages sont utilisés :

Java est le langage dans lequel le programme de gestion de MTree a été initialement conçu.



Rust est le langage dans lequel je devais réécrire le programme.

Voici les différents points qui opposent ces langages et leurs conséquences pour ma mission :

3.3.1 - Orientation

L'orientation dans un programme indique comment écrire avec ce dernier. Les manières de réaliser un algorithme dans un langage changent en fonction de l'orientation de ce langage.

Java est langage de programmation **orienté objet**. Il s'axe sur la création de structures de données appelées **'classes'**. Les classes sont des "plans de construction" qui servent à créer des **'instances'** possédant des valeurs leur appartenant (**'attributs'**) et des fonctions pour les

traiter (**'méthodes'**). Il est possible pour une classe d'étendre d'une autre, lui permettant d'avoir les mêmes attributs et méthodes que ceux de la classe originelle.

Rust est orienté sur la **programmation multi-paradigme**. Il permet aussi bien de travailler en orienté objet comme le Java que de faire de l'orienté fonctionnel. Sur ce point **Rust accorde une liberté de la manière globale de programmer**.

3.3.2 – Syntaxe

Les langages Java et Rust ont tous deux une syntaxe différente sur des éléments de base de la programmation. Par exemple : voici un morceau de code permettant d'afficher « Hello world ! » suivi du résultat de 2^8.

En Java

```
public class Exemple {
  public static void main(String args[]) {
    //Le programme demarre ici
    System.out.println("Hello word ! " + 
power(2,8));
  }

  public static int power(int x, int pow) {
    int res = 1;
    for (int i=0; i<pow; i++) {
      res *= x;
    }
    return res;
  }
}
```

En Rust

```
pub fn main() {
  //Le programme démarre ici
  println!("Hello world ! {}",power(2,8));
}

pub fn power(x: i32, pow: i32) -> i32 {
  let mut res = 1;
  for i in 0..pow {
    res *= x;
  }
  res
}
```

Les différences majeures que l'on peut noter :

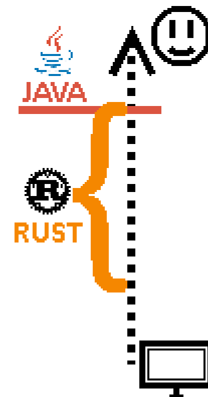
En Java, on doit déclarer des fonctions dans des classes. En Rust, ce n'est pas obligatoire.

En Java, on doit indiquer le type d'une variable avant son nom. En Rust, cela se fait entre le nom et la valeur. Dans certains cas on n'a pas besoin de préciser le type de variable en Rust, car ce dernier arrive à deviner de quel type est la variable. C'est l'inférence de type.

En Java on peut faire des boucles for avec 3 instructions : initialisation, condition d'arrêt et étape répétée. En Rust, ce genre de boucle n'existe pas directement. Il faut passer par une boucle for qui parcourra une liste de valeurs. Pour créer une liste contenant des entiers de X à Y-1, il faut suffit d'écrire « X..Y ».

3.3.3 - Niveau de langage

Le niveau d'un langage indique pour qui il est le plus facilement compréhensible. **Un langage de haut niveau indique que ce dernier est proche de ce que comprend l'humain**, rendant la programmation avec facile. À l'inverse, **un langage bas niveau est plus proche d'une machine**. Il perd en facilité de compréhension, mais est en échange plus facile à être exploiter par la machine, et donc plus rapide.



Java est un langage plutôt haut niveau. Il est donc facile à comprendre pour les humains, moins par les machines.

À l'inverse, **Rust est un langage pouvant être haut comme bas niveau**. Il est compréhensible par l'homme, mais possède des aspects permettant de faire des optimisations bas niveaux comme une gestion précise de la mémoire sur le processeur.

3.3.4 - Gestion de la mémoire

En informatique, gérer la mémoire est très important. Chaque fonction exécutée utilise de la mémoire allouée, déplacée, transformée et libérée. Pour le X-Match, cette gestion est essentielle car ce service traite un nombre important de données et peut être exploité par plusieurs utilisateurs simultanés sur un même serveur. Il faut non seulement avoir l'algorithme le plus rapide, mais également réduire la mémoire exploitée et limiter les problèmes de saturation de données, pouvant entrainer l'incapacité d'allouer de la mémoire, et avec l'arrêt prématuré du programme.

Java gère la mémoire avec un **Garbage Collector** (ramasse-miette). Cet algorithme tourne en parallèle du logiciel et vérifie ce que ce dernier n'utilise plus et libère la mémoire qui y était alloué. Ce principe permet de ne pas s'inquiéter du fonctionnement de la mémoire, mais ne permet pas de la gérer purement et peut entrainer des fuites de mémoire (mémoire allouée inutilisée). De surcroit, Le Garbage Collector est un algorithme qui tourne en parallèle du programme. Ce qui implique que **du temps est consacré à la recherche de mémoire à libérer**, alors qu'il pourrait être alloué à l'exécution d'un processus.

Rust gère la mémoire avec le paterne RAII (Resource Acquisition Is Initialization, paterne repris du C++). Les variables sont gérées pour exister d'une instruction précise du programme

à une autre. Pour être sûr de ce comportement, Rust ajoute les notions de **Ownership** (appartenance) et **Lifetime** (temps de vie).

Ownership : Dans les structures de données, une instance appartenant à une autre disparaitra en même temps qu'elle, à l'exception de si l'instance est déplacée ailleurs. Mais une instance n'aura toujours qu'un seul et unique propriétaire. Il est tout de même possible de créer des pointeurs, des références vers une variable afin qu'elle puisse être utilisée ailleurs, mais en respectant des règles précises. Ce référencement est appelé **Borrowing** (prêt).

Lifetime : Parfois, dans les fonctions, on doit ressortir un pointeur. Parfois, ce pointeur doit être un des pointeurs saisi en paramètre de la fonction. De base, Rust n'accepte pas ce type de fonction simplement. Il faut préciser que le pointeur sorti vit aussi longtemps que les pointeurs en entrée pour que la fonction soit valide. Pour donner cette précision, il faut faire appel au lifetime.

Pour résumer : **Rust est très stricte sur la mémoire**. Cela permet de libérer cette dernière dès qu'on ne s'en sert plus. En contrepartie, il faut faire preuve de beaucoup de rigueur lorsque l'on programme, et éviter au maximum de référencer une même donnée plusieurs fois dans des structures différentes. Sinon on risque d'avoir des problèmes pour compiler le programme.

Enfin, Rust considère que toutes les données dont la taille est connue à la compilation sont dans la pile, à l'exception de celles balisées dans des **Box** qui sont alors dans le tas. À l'inverse, Java considère que toute instance d'une classe est dans le tas. L'intérêt de la pile et du tas est le suivant : les données dans la pile ont une taille définie et fixe. On sait exactement où elles se trouvent et on peut les retrouver directement. Dans le tas, on ne sait pas où les retrouver directement et il faut passer par une étape avant de savoir où elles sont, mais il est possible de faire varier leur taille.

3.3.5 - Compilation

La compilation est l'étape qui permet de **convertir un programme écrit en quelque chose que la machine puisse exploiter**, le langage machine.

La compilation de programme Java ne crée pas de logiciel directement exécutable par la machine, mais **un programme en bytecode qui doit être exploité par un algorithme tiers** : la **machine virtuelle Java**. Pour le X-Match, cela impose de soit ralentir le service en démarrant la machine virtuelle à chaque appel, soit d'avoir une machine virtuelle tournant en

permanence sur l’ordinateur (type d’exécution de logiciel appelé démon) et risquer alors de cumuler les fuites de mémoire, ce qui peut poser des problèmes au serveur hébergeant le service.

Les programmes Rust compilés sont **directement exploitables par la machine**, ce qui permet de rendre les programmes utilisable sans intermédiaire, sans devoir démarrer au préalable un autre logiciel. De plus, Rust peut être compilé en WebAssembly, ce qui le rend exploitable facilement sur navigateur web, le domaine dans lequel est réalisé le X-Match.

3.3.6 - Autres différences

Le **Ownership** de Rust indique aussi la possibilité de modifier des données. **Seul le propriétaire d'une donnée peut la modifier, à moins de passer par une référence mutable**. Mais il ne peut y avoir qu'un et un seul endroit du programme où la donnée peut être modifiée à la fois. Cela permet de **gérer l'exécution d'algorithmes parallèles d'un même programme (threads) sans risquer d'erreurs de modification** (deux algorithmes tentent de modifier une variable en même temps).

Rust ne possède pas de valeur Null contrairement à de nombreux autres langages comme Java. Dès qu'une variable existe, elle doit avoir une valeur. Cependant, Rust permet d'imiter le Null avec une énumération, une structure venant de la programmation fonctionnelle pouvant avoir plusieurs formes, dont chaque forme possède différents attributs. L'énumération en question est "**Option**" qui possède la forme "**Some(valeur)**" qui contient la valeur de l'objet et "**None**" qui est l'équivalent du Null. Pour exploiter les énumérations, on peut passer par une instruction "match" qui vérifie avant si on a Some ou None. Mais comme on doit souvent renvoyer une erreur dans le cas d'un None (car on est sûr d'avoir un Some à cet endroit de l'algorithme), Option possède la méthode "unwrap()" qui permet de renvoyer directement la valeur contenue, ou de lancer une erreur si on a None.

Enfin, Rust considère que **les variables sont par défaut fixes**, qu'elles ne peuvent pas être modifiées, là où Java considère que toute variable peut être modifiée. Pour indiquer à Rust que quelque chose est modifiable, il faut le déclarer avec le mot clef "**mut**" (mutable).

3.3.7 - Résumé

Voici un tableau permettant de résumer les différences entre Java et Rust :

Point de comparaison	JAVA	RUST
Orientation	Orienté objet	Multi-paradigme
Niveau de langage	Haut niveau	Haut comme bas niveau
Gestion de la mémoire	Permissive et invisible Tout objet est dans le tas	Très stricte et contraignante mais économe, optimisée et prédictible Utiliser des Box pour mettre un objet dans le tas
Compilation	Besoin d'un logiciel tiers pour exécuter le code	Directement exploitable par la machine, compatible en WebAssembly
Autre	Valeur Null Variables modifiables de base Tout le monde peut modifier un même élément Variable mutable de base	Option Some(valeur)/None Besoin du mot clef "mut" pour altérer Un seul peut modifier à la fois Besoin du mot clef "mut" pour rendre mutable

3.4 - Coordonnées et distances

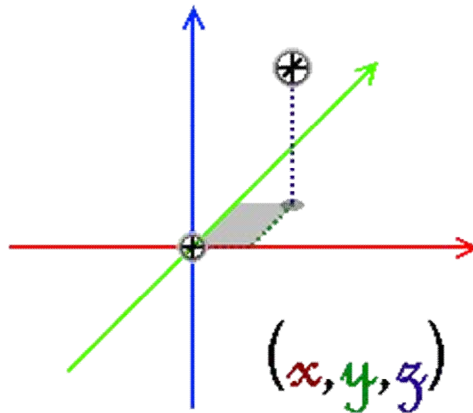
Pour faire fonctionner le M-Tree, la base est la notion de distance. C'est elle qui indique à la structure si les coordonnées sont comprises dans un nœud ou dans l'autre en comparant la distance entre le centre du nœud et le point au rayon du nœud.

Pour un même algorithme de M-Tree, on peut choisir des coordonnées et une méthode de calcul de distances qui peuvent faire fonctionner le M-Tree différemment.

Le programme Java initial avait trois types de coordonnées et quatre méthodes de calcul de distance compatibles avec le M-Tree.

3.4.1 - Coordonnées 3D

Les coordonnées en trois dimensions sont les plus basiques : on prend un espace et on lui associe trois vecteurs : $\vec{x}$, $\vec{y}$ et $\vec{z}$. Pour chaque point de l'espace, il existe une seule et unique combinaison de réels a, b et c tels que $a \times \vec{x} + b \times \vec{y} + c \times \vec{z}$ mène à ce point.



3.4.1.a : Représentation graphique de coordonnées 3D

Pour ce type de coordonnées, il existe deux types de distances dans le programme Java.

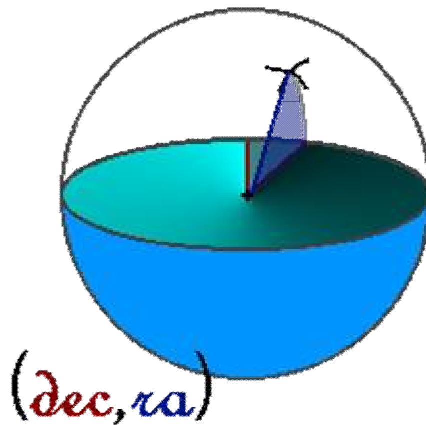
La distance Euclidienne est la plus classique. Pour de points A et B, la distance entre ces deux points est la longueur du vecteur qui sépare ces deux points. Si ce vecteur est écrit $a \times \vec{x} + b \times \vec{y} + c \times \vec{z}$, alors la distance entre A et B est $\sqrt{a^2 + b^2 + c^2}$. Si A a pour coordonnées $a_1 \times \vec{x} + a_2 \times \vec{y} + a_3 \times \vec{z}$ et B a pour coordonnées $b_1 \times \vec{x} + b_2 \times \vec{y} + b_3 \times \vec{z}$, alors la distance entre A et B est égale à $\sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + (a_3 - b_3)^2}$.

La distance Sphérique est plus proche de ce qu'utilise l'observatoire. Elle retranscrit des coordonnées en trois dimensions en des coordonnées équatoriales (voir 3.4.2).

3.4.2 - Coordonnées équatoriales

Les coordonnées équatoriales représentent des points sur la sphère unité (de rayon 1) munie d'un équateur et d'une origine sur cet équateur. C'est la méthode la plus courante pour représenter les points dans les bases de données astronomiques.

Pour un point de l'espace, on associe deux coordonnées dec (déclinaison) [-90°;+90°] et ra (right ascension, ascension droite) [0°;360°]. dec représente l'angle séparant le point de l'équateur, tandis que ra représente l'angle entre l'axe où est décrit dec et l'origine de l'équateur.



3.4.2.a : Représentation graphique de coordonnées équatoriales

L'une des méthodes de calcul de distance entre deux points de coordonnées équatoriales est celle de Haversine. Son avantage est qu'elle est très précise sur de petites distances, comme c'est le cas avec les nombreux astres pouvant être très proches dans le ciel. Elle est en revanche moins précise pour de grandes distances (comme deux points opposés sur la sphère).

3.4.3 - Chaines de caractères.

Les chaines de caractères peuvent être assimilées à des coordonnées. Pour calculer leur distance, le programme Java utilisait la méthode de Levenshtein :

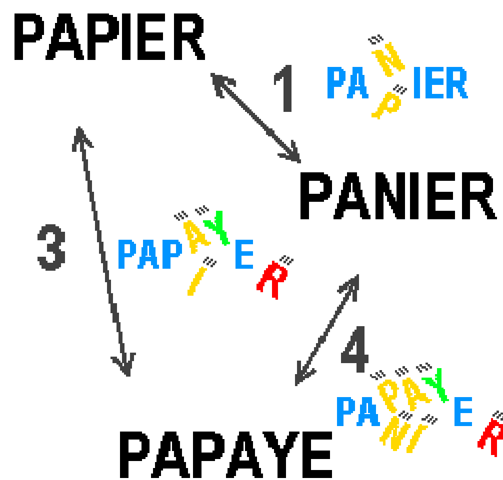
Soit trois opérations sur les chaines :

On remplace un caractère par un autre.

On supprime un caractère.

On rajoute un caractère.

La distance de Levenshtein entre deux chaines correspond au nombre minimum d'opérations que l'on doit effectuer pour passer d'une chaine à l'autre.



3.4.3.a : Représentation de la distance de Levenshtein

3.5 - Le M-Tree dans X-Match

Dans X-Match, le MTree est utilisé de la façon suivante :

- On place une base de données (catalogue) (entière ou une partie) dans un MTree.
- On prend le second catalogue et, pour chaque élément, on recherche les éléments les plus proches autour des coordonnées de ce second élément.

Là où le MTree a son avantage, c'est qu'il permet de facilement implémenter les rayons et des rayons variant avec le temps. En effet, les étoiles étant mobiles, leur position peut changer en fonction de la date. Et plus généralement il est possible d'avoir des imprécisions lorsque l'on calcule les coordonnées d'un astre. Pour être sûr de prendre toutes les étoiles pouvant être à un endroit donné à une date donnée, on peut implémenter dans le MTree la gestion de rayons variant avec le temps. On peut aussi stocker l'erreur associée aux coordonnées en faisant plus simplement des données à rayon.

Le programme Java possède 3 variantes de MTree :

- Le **MTree** classique, permettant de stocker des points.
- Le **MTreeR** (Radius) pour stocker des sphères. Ces sphères peuvent correspondre à des imprécisions de calculs de coordonnées d'une étoile.
- Le **MTreeTR** (Timed Radius) pour stocker des sphères dont le rayon varie en fonction du temps. Cela permet de croiser les bases de données proprement même si les

données ont des dates différentes, en identifiant dans quel périmètre une étoile pourrait se trouver.

4 - Le déroulement du stage

4.1 - Les conditions de travail

4.1.1 - Conditions globales

Mon stage a duré 10 semaines. En pratique, il y a eu de nombreux jours fériés et plusieurs absences de ma part justifiées par des recherches dans le cadre de mes poursuites d'études.

Mon poste de travail se situait dans la bibliothèque. Cette dernière était très peu utilisée du fait que de nombreux ouvrages sont désormais numérisés. Nous étions six stagiaires au total dont quatre autour de la même table.

Nous étions rarement dérangés. Parfois quelqu'un traversait la bibliothèque pour accéder à une autre salle, d'autres fois des visiteurs venaient pour admirer le télescope situé au milieu de la bibliothèque, mais le plus souvent nous avions la visite de nos tuteurs pour regarder notre travail et nous aider à avancer.

Mon maitre de stage se trouvait dans le même bâtiment. Cela me permettait d'aller le voir pour lui demander de l'aide, des informations ou des précisions, pour le tenir au courant des résultats.

4.1.2 - Outils de travail

Je disposais d'un poste de travail fixe tournant sous un système d'exploitation Linux connecté au réseau de l'ObAS. Je pouvais également utiliser mon ordinateur personnel comme second support.

Comme tous les stagiaires, j'avais un dépôt GIT pour placer mon projet en fonction de son avancement. Je pouvais également écrire dans le wiki interne de l'ObAS mon avancement quotidien. Cela permettait de pouvoir revoir ce que j'avais fait, m'aidant notamment pour écrire mon rapport de stage, en plus de pouvoir renseigner de potentiels futurs stagiaires qui traiteraient un sujet similaire au miens sur mes façons de résoudre les difficultés rencontrées.

En plus de ces outils, j'ai utilisé Visual Studio Code, une IDE permettant de facilement utiliser Rust après quelques commandes, ainsi que Cargo, un utilitaire servant à gérer des projets Rust. Visual Studio Code permet également de gérer des projets Java, ce qui m’a permis de visualiser et de faire tourner les deux en parallèle dans des conditions similaires, renforçant l'exactitude des tests de performance.

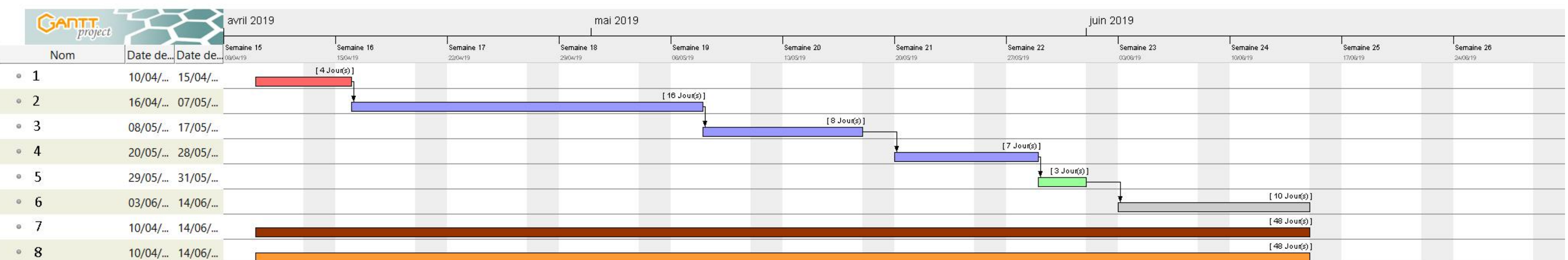
Pour les tests de performances (ou benchmarking), j'ai utilisé du côté de Java les algorithmes de tests que mon tuteur avait déjà réalisés. Du côté de Rust, j'ai ajouté la bibliothèque Criterion qui permet de faire du benchmarking simplement et avec des résultats significatifs et détaillés.

4.2 - L'avancement du projet étape par étape

Grâce au wiki, je suis en mesure de diviser mon stage en plusieurs étapes, chacune s'articulant autour d'une partie particulière de ma mission de conversion. Je vais donc présenter le déroulement de mon stage de façon chronologique.

N	Nom	Début	Fin
1	Traduction brute	**(début) 10/04**	15/04
2	Écriture d’un M-Tree de base	16/04	07/05
3	Recherches du M-Tree	09/05	17/05
4	Tests de performances et Tests Java-Rust	22/05	28/05
5	Documentation	29/05	31/05
6	Fin du stage	03/06	**14/06 (fin)**
7	Apprentissage du Rust	**(début) 10/04**	**14/06 (fin)**
8	Étude du programme Java	**(début) 10/04**	**14/06 (fin)**

4.2.b : Légende des périodes



4.2.b : Diagramme de Gantt du déroulement de mon stage

4.2.1 - Formation à Rust (continue)

Tout au long du stage, je me suis formé à Rust principalement avec le manuel disponible en ligne (voir webographie lien 1). Mais les premières semaines étaient celles où j'ai le plus appris de Rust.

4.2.2 - Étude du programme Java et du MTree (continue)

Tout en me formant à Rust, j'ai lu et appris comment fonctionnait le programme Java de gestion de MTree ainsi que le MTree en lui-même à travers le document que m'a donné mon maitre de stage (voir référence en bibliographie). Cela se faisait en continue tout au long du stage. J'étudiai les algorithmes et les réécrivais afin de bien comprendre leur comportement, et je tentais de faire des diagrammes de classe de l'architecture du programme Java pour répertorier tout ce qu'il contenait.

4.2.3 - Traduction "brute" (10/04 - 15/04)

Ma première tentative de traduire le programme fut une tentative "brute". J'ai tenté de convertir le programme de Java à Rust ligne de code par ligne de code. Mon tuteur me conseilla de commencer par traduire la classe MTreeImpl, qui sert à gérer le fonctionnement du MTree.

J'ai choisi de convertir les interfaces de Java pour les remplacer par leur meilleur homologue en Rust : les traits.

J'ai recréé un des premiers éléments nécessaires pour le M-Tree : les distances. Pour les distances des coordonnées 3D et équatoriales, la conversion fut rapide. Mais pour la distance de Levenshtein, débutant encore en Rust, je repris un algorithme déjà existant sur Internet, publié sous licence MIT (voir lien 3 de la Webographie).

Après quelques jours, mon manque d'expérience en Rust et la différence de structuration entre Java et Rust m'ont bloqué. Voici un exemple de problème qui rend difficile la traduction brute :

En Jave l'arbre est conçu de telle façon à ce que chaque nœud connaisse non seulement ses enfants, mais aussi son parent. En Rust, faire ce genre de structure où deux éléments se référence l'un l'autre n'est pas possible de façon simple à cause de la règle de l'Ownership. Pour le faire, il faudrait user de RefCell, une structure Rust permettant d'avoir plusieurs

pointeurs sur un même élément. Mais user de RefCell complexifierait le programme et mal les utiliser pourrait entrainer des fuites de mémoires.

En plus de ce problème d'autres encore m'ont fait obstacle. J'ai alors décidé de changer de méthode de programmation.

4.3.3 - Écriture d'un MTree de base (16/04 - 07/05)

J'ai décidé de reprendre la base du MTree, d'en refaire un à partir de zéro. J'y ai intégré petit à petit les fonctionnalités du programme Java une par une. Cette méthode m'a permis non seulement de plus facilement me retrouver dans la traduction et de comprendre plus aisément ce que je faisais mais également de faire une structure plus adaptée à Rust que celle présente en Java.

J'ai également changé le paterne utilisé, Java utilise un paterne composite qui fait que l'ont différencie les feuilles et les nœuds par deux classes héritant de la même structure. J'ai remplacé ce paterne par l'usage d'énumérations.

Au début j'ai voulu représenter les nœuds du MTree par une énumération à trois formes :

```
enum MTreeNode {
    Root(Box<Vec<MTreeNode>>, Box<Ball>),
    Node(Box<Vec<MTreeNode>>, Box<Ball>, Box<MTreeNode>),
    Leaf(Box<Vec<Data>>, Box<Ball>, Box<MTreeNode>)
}
```

Où Root représente une racine avec la liste des sous-nœuds enfants et une Ball (sphère) pour représenter son aire ; Node pour représenter la même chose avec une référence vers son parent et Leaf qui à la différence de Node a pour enfant des Data, le trait représentant des données.

Cette structure m'a confronté à des problèmes. Elle n'a notamment pas été des plus logiques du fait qu'elle empêchait d'avoir une feuille pour racine. J'ai donc décidé de refaire ma structure ainsi :

```
enum MTreeNode {
    Node(Vec<MTreeNode>, Ball, Option<TreeNode>),
    Leaf(Vec<Data>, Ball, Option<MTreeNode>)
}
```

La forme Root a disparu et le parent était désormais représenté par une option. Ainsi, on pouvait trouver la racine du fait qu'il s'agissait d'une MTreeNode avec pour parent None.

Au final, cette structure ne convenait toujours pas. À cause des règles de Rust qui stipulent qu'on ne peut pas avoir plusieurs propriétaires pour une valeur, je ne pouvais pas stocker les parents de cette façon.

J'ai eu deux solutions pour régler ce problème : Soit j'utilisais des RefCells, une structure permettant d'avoir plusieurs références, mais complexifiant l'usage par des procédures de vérification d'unicité d'usage qui pourraient entraîner des bugs dans le cas de multi-threading ; Soit abandonner l'idée qu'un enfant connaisse son parent et gérer cela par appels de fonctions récursives. Puisque la première n'était pas désirable et puisque je cherchais à l'éviter en passant par la décomposition en fonctionnalités, j'optai pour cette seconde méthode.

J'ai changé donc pour une architecture plus logique :

```
struct MTreeNode {
    covered_area: Ball,
    data: MTreeNodeType
}

enum MTreeNodeType {
    Node(Vec<Box<MTreeNode>>),
    Leaf(Vec<Box<SpaceData>>)
}
```

La structure permet de stocker la Ball représentant la sphère couverte et potentiellement d'autres données communes aux nœuds et feuilles tandis que le type se charge de différencier si la structure représente un Node ou une Leaf, changeant alors le type d'enfant stocké.

J'ai également remplacé le trait Data par le trait SpaceData, qui sert à lier Data, une structure quelconque pouvant contenir des données, et Coordinates, un trait représentant des coordonnées spatiales. Cela me permettait d'éviter des duplications de données inutiles dans les Balls qui ne stockent plus que des Coordinates plutôt que des Data complètes.

J'ai également eu des difficultés pour réaliser les méthodes de construction du MTree dues à ma structure bancale. Mais avec la dernière architecture présentée, ces derniers sont devenus beaucoup plus simples à mettre en place.

Ces derniers diffèrent des algorithmes Java dans leur déroulement. Java exécute une méthode du MTree qui parcoure les nœuds jusqu'à trouver une feuille qui convienne et tente d'insérer la donnée, si cela déclenche un split (une division du nœud car ce dernier a trop d'éléments), la méthode remonte de parent en parent pour insérer le nœud résultant du split jusqu'à ce qu'il n'y ait plus de split.

Dans Rust, le MTree appelle une méthode de la racine qui, si c'est un nœud, appelle la même méthode pour un de ses enfants puis analysera ce que son enfant lui retourne. S'il lui renvoie un nœud, il insère ce nœud. Si cette insertion le split, il renvoie le nœud résultant de son split. Si en revanche la méthode est appelée sur une feuille, elle tentera d'insérer la donnée et, si elle se split, renverra la feuille résultant du split.

Pour résumer, l'algorithme Java est propre à de l'objet, utilisant une boucle pour. L'algorithme que j'ai conçu est plus proche de la programmation fonctionnel, car il fait appel à une fonction récursive. Cela permet également de traiter le cas des parents sans avoir à passer par les RefCells ou par une liste qui devrait servir d'historique.

Pour l’algorithme de split, j’ai repris le même principe que Java. Cet algorithme se divise en 5 étapes :

1. On trouve les deux éléments les plus éloignés l’un de l’autre. Ces éléments sont les extrémités.
2. On sépare le reste des éléments : chacun fait partie du groupe de l’extrémité dont il est le plus proche.
3. On cherche dans chaque groupe l’élément le plus central (dont la distance maximale avec les autres éléments de son groupe est la plus petite possible).
4. On place chaque élément dans son nœud respectif. Le centre de chaque nœud est le premier élément.
5. On renvoie le nœud créé par l’algorithme pour qu’il puisse être inséré dans le parent.

Ensuite, après avoir discuté avec mon tuteur, j’ai débuté la conception une structure permettant d'instancier des M-Trees : le MTreeBuilder. Ce dernier permet de créer un MTree en lui donnant des paramètres sur plusieurs instructions distinctes.

Les paramètres en question sont :

- Le premier élément du M-Tree (qui ne doit pas être vide).
- Une structure contenant une méthode qui permet de calculer et comparer des distances entre des points.
- Un nombre maximal d'élément par nœud.

- Des structures permettant d'indiquer comment les nœuds et les feuilles doivent se splitter. Ces paramètres ont des valeurs par défaut et sont donc facultatifs. Ils sont là car, dans l'algorithme Java, il existe deux méthodes pour splitter les nœuds.

J'ai également réalisé des algorithmes de recherche basiques, pas optimisés, mais permettant d'avoir une première version fonctionnelle du MTree.

Au final, je disposai d'un algorithme de gestion de MTree opérationnel. Je l'ai vérifié avec des tests unitaires retraçant le comportement à la création, l'insertion, le split de feuille et le split de nœud. Tous semblaient justes.

4.3.4 - Recherches du MTree de base (09/05 - 17/05)

Je devais ensuite compléter le MTree afin qu'il puisse effectuer des recherches de façon plus optimisée. Celui du Range-Query était déjà sous une de ses meilleures formes. Mais les autres étaient loin de fonctionner correctement.

J'ai repris l'algorithme de recherche KNN du programme Java et l'ai implémenté dans mon MTree. Ce dernier comprend la recherche avec un rayon maximal.

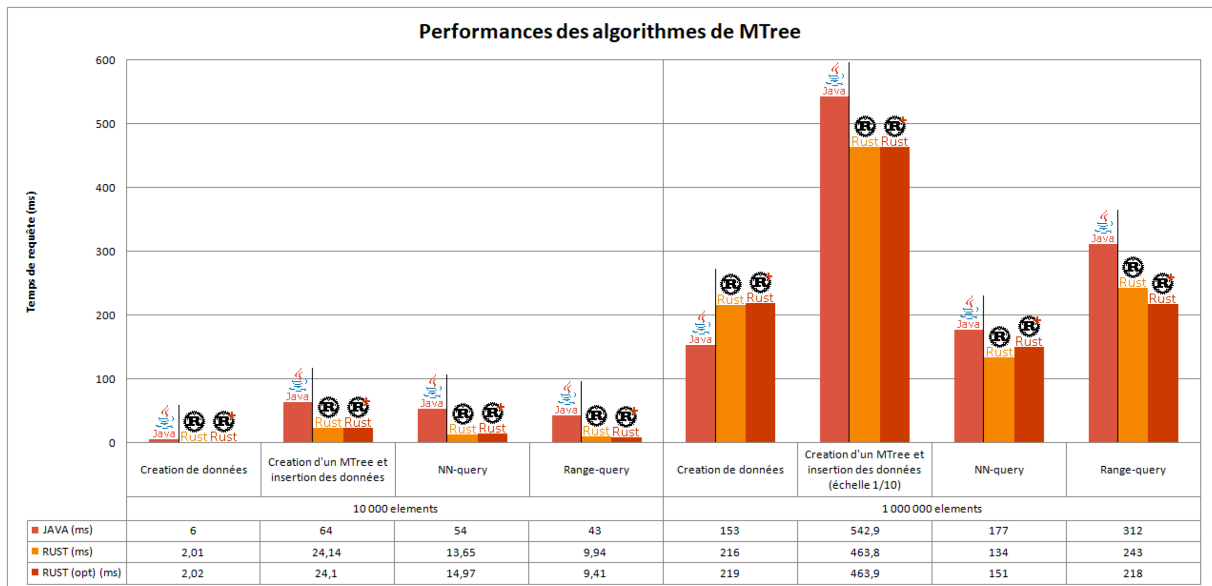
Par cet algorithme de KNN à rayon maximal, j'ai pu décliner les recherches NN et KNN en version avec et sans rayon maximal.

4.3.5 – Tests de performances et de traces Java-Rust (22/05 - 28/05)

Afin de vérifier que mon algorithme est bien plus performant que celui de Java, je devais faire du benchmarking (tests de performance). Malheureusement les solutions de benchmarking de Rust ne sont pas complètes. Pour les utiliser, il faut passer par le mode "nightly", ce qui signifie qu'il me faudrait baliser mon code comme "pas sûr" car il utiliserait des fonctionnalités "pas sûres".

Après quelques recherches, j'ai finalement trouvé Criterion, une bibliothèque permettant de faire du Benchmarking dans Rust sans passer par le mode "nightly", et de façon plus complète.

J'ai réalisé le benchmarking du côté de Java et de Rust afin d'atteindre les premiers résultats suivants.



	10 000 elements				1 000 000 elements			
	Creation de données	Creation d'un MTree et insertion des données	NN-query	Range-query	Creation de données	Creation d'un MTree et insertion des données (échelle 1/10)	NN-query	Range-query
JAVA (ms)	6	64	54	43	153	542,9	177	312
RUST (ms)	2,01	24,14	13,65	9,94	216	463,8	134	243
RUST (opt) (ms)	2,02	24,1	14,97	9,41	219	463,9	151	218

4.3.5.a : Comparaison des performances entre Java et Rust

Les tests sur des arbres à 10.000 éléments sont peu concluants à cause du principe de JIT (Just-In-Time) de Java (Java optimise le programme au fur et à mesure que ce dernier tourne). Ceux sur 1.000.000 ont déjà plus de valeur. La différence entre Rust (colonnes du milieu) et Rust+ (colonnes de droite), c'est le fait que le programme exécuté dans la colonne Rust+ est théoriquement optimisé par l'appel à une méthode d'optimisation et d'un calcul de performance adapté à l'ordinateur sur lequel Rust tourne.

La création de données est ici une génération de données aléatoires. La différence entre Rust et Java s'explique par la complexité de l'algorithme qui fournit des valeurs aléatoires. Autrement, on peut remarquer que la différence entre Rust et Java n'est pas très flagrante.

J'ai réalisé des tests pour vérifier le fonctionnement de mon MTree. Me basant sur le principe que les algorithmes Java sont justes et que mon objectif est de retourner la même chose avec Rust qu'avec Java, j'ai modifié l'algorithme Java pour qu'il me donne des fichiers contenant des données aléatoires servant à construire un MTree, puis des données aléatoires servant à tester les algorithmes de recherche et le résultat de ces recherches sur Java.

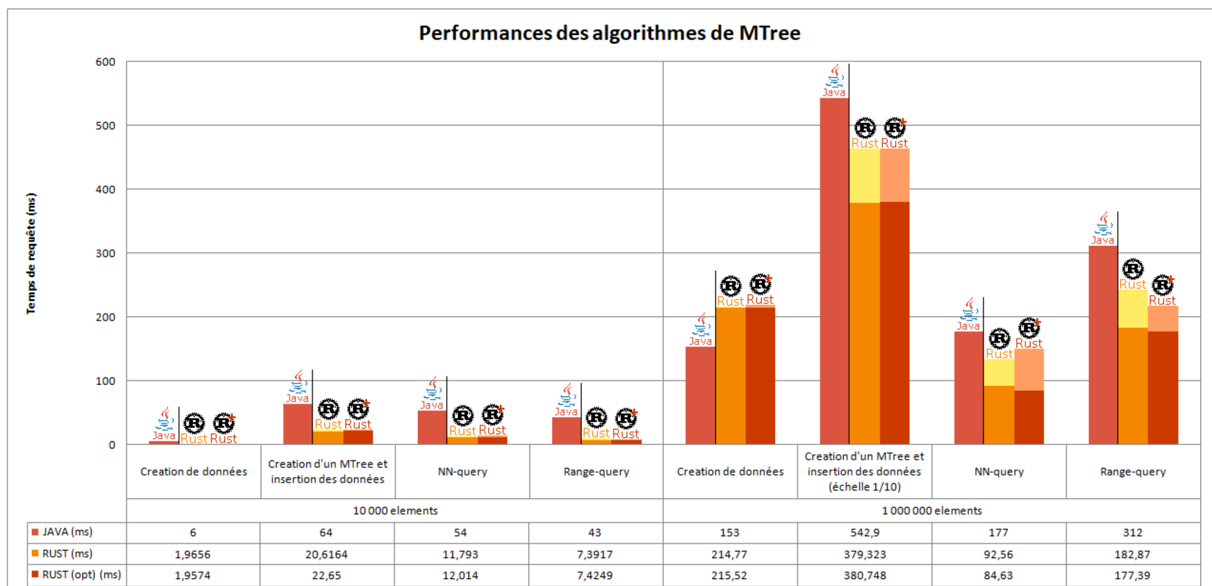
En parallèle, sur Rust, j'ai fais un algorithme qui lit ces données, construit l'arbre et vérifie ce qu'il retourne avec les recherches par rapport aux retours de Java. Cette méthode me permit de

constater des erreurs. Des données pourtant dans l'arbre, qui coïncident avec les recherches, ne fonctionnaient pas.

Pour bien vérifier d'où venait cette erreur et corriger de façon optimale, j’ai utilisé un système de tests vu en cours : les traces. J’ai créé une liste de 10.000 données à insérer dans le MTree. Dans Java, j’ai fais pour chaque insertion un fichier de sortie, chaque fichier étant une trace. Dans Rust, j’ai fais pour chaque insertion un même fichier et comparer les deux. S'il y avait une différence, Rust ne continuait pas le test et le considérait comme échoué. Je pouvais alors regarder les dernières traces pour observer les différences et voir les erreurs.

De cette façon j’ai corrigé tous mes problèmes, réglant les algorithmes d'insertion et de split qui comportaient des erreurs que mes tests unitaires ne permettaient pas de voir.

Après ces correctifs qui me permettaient d'avoir un MTree plus optimisé, j’ai refais les tests de benchmarking :



	10 000 elements				1 000 000 elements			
	Creation de données	Creation d'un MTree et insertion des données	NN-query	Range-query	Creation de données	Creation d'un MTree et insertion des données (échelle 1/10)	NN-query	Range-query
JAVA (ms)	6	64	54	43	153	542,9	177	312
RUST (ms)	1,9656	20,6164	11,793	7,3917	214,77	379,323	92,56	182,87
RUST (opt) (ms)	1,9574	22,65	12,014	7,4249	215,52	380,748	84,63	177,39

4.3.5.b : Performances des algorithmes après corrections

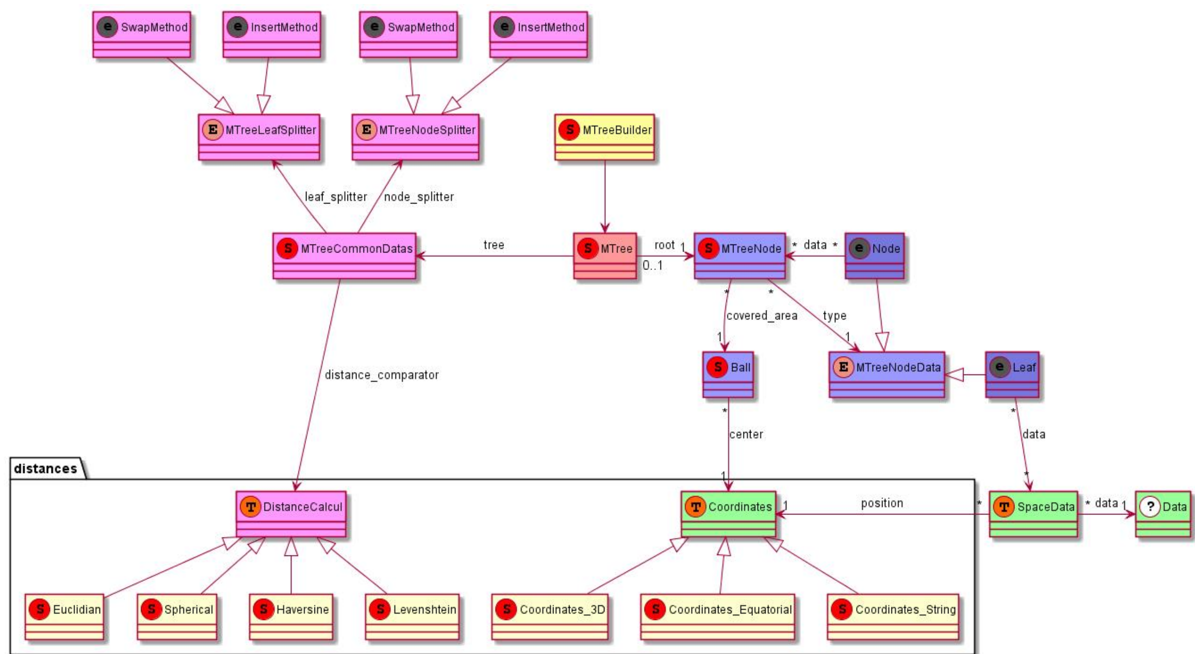
On peut voir la réduction du temps, permettant même aux algorithmes de recherche de plus proche voisin d'être presque deux fois plus rapides en Rust qu'en Java.

4.3.6 - Documentation (29/05 - 31/05)

Arrivant vers la fin du stage, j’ai décidé avant de programmer la suite de documenter mes travaux. Comme me le demanda mon maitre de stage, j’ai réalisé la documentation intégralement en anglais. J'y ai également placé du code servant à la fois d'exemple et à la fois de tests unitaires.

La documentation de Rust se fait en Markdown, un langage utilisé dans les dépôts Git dans des fichiers « .md ». Cargo m’a ensuite permis de générer une suite de fichiers html formant une documentation compatible avec le web. Je place une partie de cette documentation en annexe pour donner une idée de ce que Rust génère.

Avec cela j'ai réalisé un diagramme de classe pour représenter la structure finale de mon travail :



4.3.6.a : Diagramme de classe du programme Rust de gestion de MTree

	En rouge est représenté le MTree, la classe centrale de la gestion du projet.
	En magenta est la partie utilitaire du MTree, à savoir des données dont chaque nœud pourrait avoir besoin et qui est donc transféré de nœud en nœud à chaque appel de méthode récursif.
	En bleu sont les nœuds et les structures et énumérations liées aux nœuds.
	En vert sont les classes de base de la gestion du MTree : les coordonnées et Data, qui peut être n'importe quoi.
	En jaune est le MTreeBuilder, la classe nécessaire pour la création du MTree.
	En jaune pâle sont les déclinaisons des systèmes de coordonnées et méthodes de calcul de distances de ces coordonnées décrits dans Java et reproduits en Rust.

4.3.7 – La fin du stage (03/06 – 14/06)

J'ai consacré mes dernières semaines à la réalisation de mon rapport de stage et à la préparation de la présentation orale de mes dix semaines.

J'ai tout de même eu le temps de faire un autre M-Tree, le M-Tree R, dont le code ressemble beaucoup à celui de base. Ce dernier permet de gérer des sphères au lieu de simples points.

À l'origine je voulais intégrer la gestion du M-Tree R dans le M-Tree. Mais en étudiant bien le fonctionnement, non seulement adapter le M-Tree pour qu'il comporte le M-Tree R ralentissait l'algorithme (diminution du gain de performances par rapport à Java de 5%), mais

de plus les algorithmes de recherches étaient très différents (le M-Tree voit son rayon maximal comme une distance, le M-Tree R comme une proportion de distance divisée par la somme des rayons de l'élément recherché et du centre). J'ai donc opté pour la séparation du M-Tree R en une structure distincte du M-Tree.

Faire le M-Tree R ne me demanda que de copier-coller le code du M-Tree et de modifier certains points.

J'ai également corrigé et renforcé ma documentation, corrigé une potentielle source de bug et débuté la rédaction de tests Java-Rust pour tester le M-Tree R.

5 - La conclusion

Ces 10 semaines de stages m'ont été fortement bénéfiques. Elles m'ont permis de me confronter à un véritable projet avec des contraintes de temps et un besoin d'autonomie. Elles m'ont également permis d'apprendre un nouveau langage, le Rust, qui pourra m'être utile à l'avenir, et de m’intéresser à divers notions que je ne connaissais pas avant, comme le paterne RAII, la structure de données M-Tree ou encore les divers méthodes de calcul de distances.

Le résultat du projet sera à terme utilisé. Il ne le sera cependant pas directement du fait que je n’ai pas implémenté entièrement le MTreeR ni le MTreeTR. Mais il pourra être repris sans autre difficulté que l’apprentissage de Rust. J'ai pris soin de documenter mon code lors des dernières semaines pour permettre une reprise facile.

En plus de devoir ajouter la gestion du MTreeTR et vérifier le fonctionnement du MTreeR, il faut effectuer le reste de code permettant de l'intégrer entièrement à X-Match et ajouter la gestion multi-threading pour la création du MTree. Ensuite, il est possible d’implémenter les algorithmes de suppression déjà écrits en Java. Ceux-ci ne sont pas des plus importants pour le X-Match, mais pourraient avoir leur utilité dans d’autres usages du M-Tree. Enfin, il est peut-être possible d’améliorer les performances des algorithmes en utilisant des éléments plus complexes de Rust.

6 - Bibliographie & Webographie

6.1 Bibliographie

M-tree : An Efficient Access Method for Similarity Search in Metric Spaces

de Paolo Ciaccia, Marco Patella et Pavel Zezula ; pages 426 - 435

6.2 Webographie

1) Site officiel de Rust :

https://www.rust-lang.org/

2) Manuel en ligne de Rust :

https://doc.rust-lang.org/book/foreword.html

3) Algorithme de Levenshtein de T. Wormer :

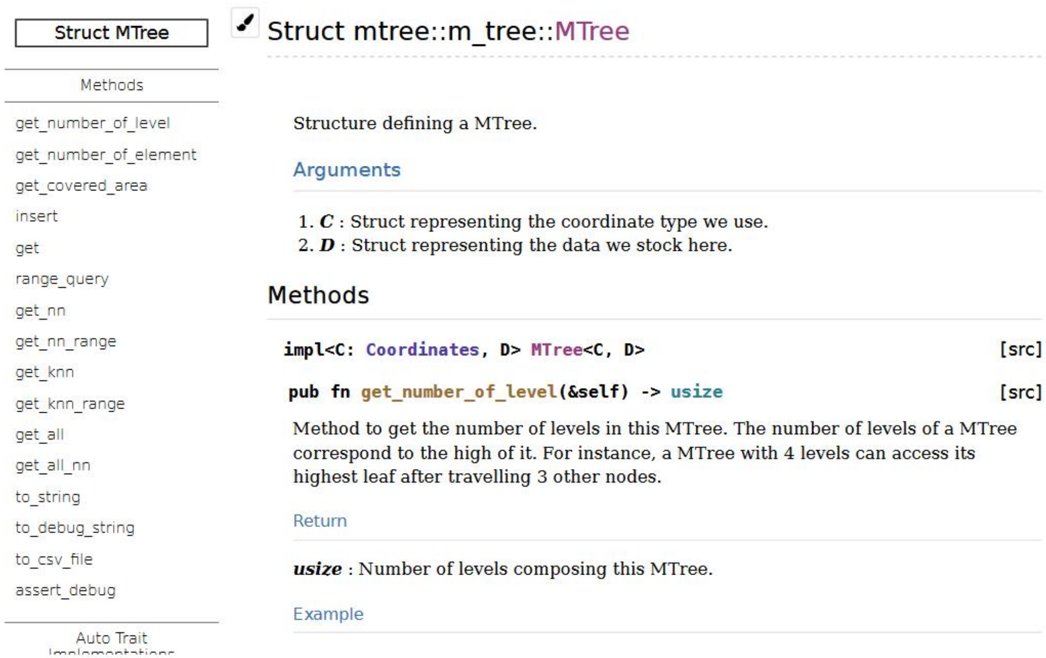
https://github.com/wooorm/levenshtein-rs/blob/src/lib.rs

4) Page Wikipedia sur le M-Tree (en anglais) :

https://en.wikipedia.org/wiki/M-tree

7 – Annexes

Les annexes se composent de 5 images de la documentation générée par Cargo sur le MTree.



Struct MTree

Methods

get_number_of_level
get_number_of_element
get_covered_area
insert
get
range_query
get_nn
get_nn_range
get_knn
get_knn_range
get_all
get_all_nn
to_string
to_debug_string
to_csv_file
assert_debug

Auto Trait Implementations

Struct mtree::m_tree::MTree

Structure defining a MTree.

Arguments

1. *C* : Struct representing the coordinate type we use.
2. *D* : Struct representing the data we stock here.

Methods

```
impl<C: Coordinates, D> MTree<C, D>                        [src]
pub fn get_number_of_level(&self) -> usize                  [src]
```

Method to get the number of levels in this MTree. The number of levels of a MTree correspond to the high of it. For instance, a MTree with 4 levels can access its highest leaf after travelling 3 other nodes.

Return

usize : Number of levels composing this MTree.

Example

Annexe 1/5 : Page 1 de la documentation du MTree

Struct MTree

Methods

get_number_of_level
get_number_of_element
get_covered_area
insert
get
range_query
get_nn
get_nn_range
get_knn
get_knn_range
get_all
get_all_nn
to_string
to_debug_string
to_csv_file
assert_debug

Auto Trait Implementations

```
// import of basic structs for testing and examples
extern crate mtree;
use mtree::tests::*;
use mtree::basic_distances::coordinates_3d::*;

// creation of a basic MTree with an xyz-Euclidean space, it has a maxi
let mut mtree = create_default_mtree(TestSpaceData::new_xyz(2.,0.,0.,St

// insertion of some data
mtree.insert(Box::new(TestSpaceData::new_xyz(1.5,0.,0.,String::from("1"
mtree.insert(Box::new(TestSpaceData::new_xyz(2.,0.,0.,String::from("2")

//the mtree has still 1 level : the root
assert_eq!(mtree.get_number_of_level(),1);

//insertion of a method that will rise a new level
mtree.insert(Box::new(TestSpaceData::new_xyz(-50.,3.,0.,String::from("-

//the mtree has now 2 levels : the root level and the leaves level
assert_eq!(mtree.get_number_of_level(),2);
```

```
pub fn get_number_of_element(&self) -> usize                [src]
```

Method to get the total amount of elements in this MTree. It correspond to the total amount of SpaceData stocked in this MTree.

Annexe 2/5 : Page 2 de la documentation du MTree

Struct MTree

Methods

get_number_of_level
get_number_of_element
get_covered_area
insert
get
range_query
get_nn
get_nn_range
get_knn
get_knn_range
get_all
get_all_nn
to_string
to_debug_string
to_csv_file
assert_debug

Auto Trait

Return

usize : Total number of SpaceData in this MTree.

Example

```
// import of basic structs for testing and examples
extern crate mtree;
use mtree::tests::*;
use mtree::basic_distances::coordinates_3d::*;

// creation of a basic MTree with an xyz-Euclidean space
let mut mtree = create_default_mtree(TestSpaceData::new_xyz(2.,0.,0.,St

// insertion of some data
mtree.insert(Box::new(TestSpaceData::new_xyz(1.5,0.,0.,String::from("1"
mtree.insert(Box::new(TestSpaceData::new_xyz(2.,0.,0.,String::from("2")
mtree.insert(Box::new(TestSpaceData::new_xyz(-50.,3.,0.,String::from("-

//the mtree has now 4 elements in itself
assert_eq!(mtree.get_number_of_element(),4);
```

```
pub fn get_covered_area(&self) -> &Ball<C>
```
[src]

Method to get the covered area of this MTree. Every node, leaf and SpaceData of this MTree is contained within this area.

Annexe 3/5 : Page 3 de la documentation du MTree

Struct MTree

Methods

get_number_of_level
get_number_of_element
get_covered_area
insert
get
range_query
get_nn
get_nn_range
get_knn
get_knn_range
get_all
get_all_nn
to_string
to_debug_string
to_csv_file
assert_debug

Auto Trait

Return

&Ball : A ball represeting the entire covered area of this MTree.

Example

```
// import of basic structs for testing and examples
extern crate mtree;
use mtree::tests::*;
use mtree::basic_distances::coordinates_3d::*;

// creation of a basic MTree with an xyz-Euclidean space
let mut mtree = create_default_mtree(TestSpaceData::new_xyz(2.,0.,0.,St

// insertion of some data
mtree.insert(Box::new(TestSpaceData::new_xyz(1.5,0.,0.,String::from("1.

//now the mtree covers a ball with center the first point and radius th
let covered_area = mtree.get_covered_area();
assert_eq!(covered_area.center.x,2.);
assert_eq!(covered_area.center.y,0.);
assert_eq!(covered_area.center.z,0.);
assert_eq!(covered_area.radius,0.5);
```

```
pub fn insert(&mut self, point: Box<dyn SpaceData<C, D>>)
```
[src]

Annexe 4/5 : Page 4 de la documentation du MTree

Struct MTree

Methods

get_number_of_level
get_number_of_element
get_covered_area
insert
get
range_query
get_nn
get_nn_range
get_knn
get_knn_range
get_all
get_all_nn
to_string
to_debug_string
to_csv_file
assert_debug

Auto Trait Implementations

Method to insert into the MTree a new piece of data.

Arguments

1. **point** : ***Box<SpaceData<C,D>>*** : data to insert into this MTree.

Example

Annexe 5/5 : Page 5 de la documentation du MTree