CentraleSupélec

CentraleSupélec

Geometric Methods in Data Analysis

Final Project

Search Algorithms in Metric Trees

Authors:
Antoine Pupin
Timothee Samou

Teacher:
Frederic Cazals

February 26, 2017

Contents

1 Introduction

The goal of this project is to carry out experiments on the performances of search procedures for metric trees. We are going to implement metric trees with several build and search strategies. We are also going to extend this application on more complex data that are molecular conformations.

We created a git repository for this project and the git is available through the following link : https://github.com/timothee001/MetricTree

2 Technical environment

The project has been developed using C++ with Visual Studio Ultimate 2015

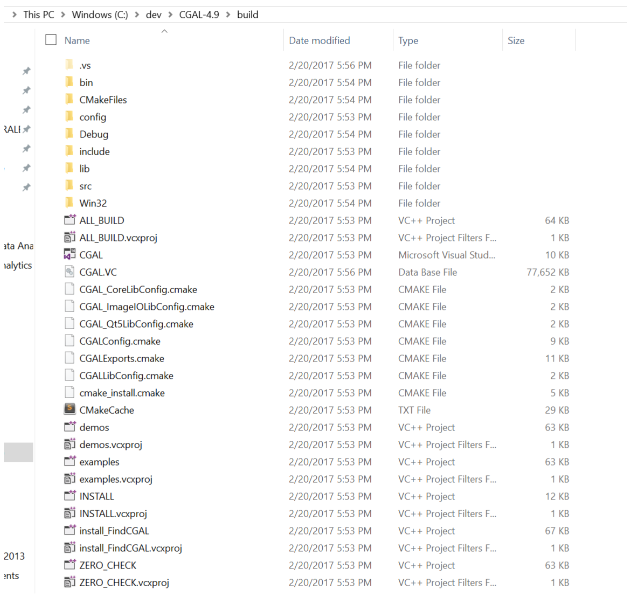
All the libraries that are needed in order to do conformations part have been fully downloaded, compiled and integrated in the project directly with CMAKE.

2.1 CGAL and SBL

Two libraries were needed to implement the metric tree search implementation on conformation, CGAL and SBL.

"CGAL is a software project that provides easy access to efficient and reliable geometric algorithms in the form of a C++ library. CGAL is used in various areas needing geometric computation, such as geographic information systems, computer aided design, molecular biology, medical imaging, computer graphics, and robotics."

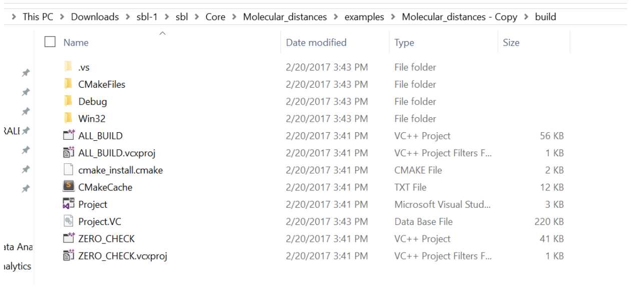
"The Structural Bioinformatics Library (SBL) is a Template C++/Python library for solving structural biology problems. The SBL provides programs (executables) for end-users and a rich framework todevelop novel applications."



› This PC › Windows (C:) › dev › CGAL-4.9 › build

Name	Date modified	Type	Size
.vs	2/20/2017 5:56 PM	File folder	
bin	2/20/2017 5:54 PM	File folder	
CMakeFiles	2/20/2017 5:54 PM	File folder	
config	2/20/2017 5:53 PM	File folder	
Debug	2/20/2017 5:54 PM	File folder	
include	2/20/2017 5:53 PM	File folder	
lib	2/20/2017 5:54 PM	File folder	
src	2/20/2017 5:53 PM	File folder	
Win32	2/20/2017 5:54 PM	File folder	
ALL_BUILD	2/20/2017 5:53 PM	VC++ Project	64 KB
ALL_BUILD.vcxproj	2/20/2017 5:53 PM	VC++ Project Filters F...	1 KB
CGAL	2/20/2017 5:53 PM	Microsoft Visual Stud...	10 KB
CGAL.VC	2/20/2017 5:56 PM	Data Base File	77,652 KB
CGAL_CoreLibConfig.cmake	2/20/2017 5:53 PM	CMAKE File	2 KB
CGAL_ImageIOLibConfig.cmake	2/20/2017 5:53 PM	CMAKE File	2 KB
CGAL_Qt5LibConfig.cmake	2/20/2017 5:53 PM	CMAKE File	2 KB
CGALConfig.cmake	2/20/2017 5:53 PM	CMAKE File	9 KB
CGALExports.cmake	2/20/2017 5:53 PM	CMAKE File	11 KB
CGALLibConfig.cmake	2/20/2017 5:53 PM	CMAKE File	2 KB
cmake_install.cmake	2/20/2017 5:53 PM	CMAKE File	5 KB
CMakeCache	2/20/2017 5:53 PM	TXT File	29 KB
demos	2/20/2017 5:53 PM	VC++ Project	63 KB
demos.vcxproj	2/20/2017 5:53 PM	VC++ Project Filters F...	1 KB
examples	2/20/2017 5:53 PM	VC++ Project	63 KB
examples.vcxproj	2/20/2017 5:53 PM	VC++ Project Filters F...	1 KB
INSTALL	2/20/2017 5:53 PM	VC++ Project	12 KB
INSTALL.vcxproj	2/20/2017 5:53 PM	VC++ Project Filters F...	1 KB
install_FindCGAL	2/20/2017 5:53 PM	VC++ Project	67 KB
install_FindCGAL.vcxproj	2/20/2017 5:53 PM	VC++ Project Filters F...	1 KB
ZERO_CHECK	2/20/2017 5:53 PM	VC++ Project	63 KB
ZERO_CHECK.vcxproj	2/20/2017 5:53 PM	VC++ Project Filters F...	1 KB

Figure 1: Compiled CGAL Library for VS

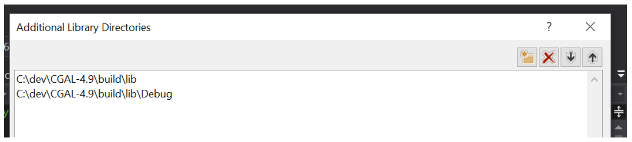


› This PC › Downloads › sbl-1 › sbl › Core › Molecular_distances › examples › Molecular_distances - Copy › build

Name	Date modified	Type	Size
.vs	2/20/2017 3:43 PM	File folder	
CMakeFiles	2/20/2017 3:43 PM	File folder	
Debug	2/20/2017 3:43 PM	File folder	
Win32	2/20/2017 3:43 PM	File folder	
ALL_BUILD	2/20/2017 3:41 PM	VC++ Project	56 KB
ALL_BUILD.vcxproj	2/20/2017 3:41 PM	VC++ Project Filters F...	1 KB
cmake_install.cmake	2/20/2017 3:41 PM	CMAKE File	2 KB
CMakeCache	2/20/2017 3:41 PM	TXT File	12 KB
Project	2/20/2017 3:41 PM	Microsoft Visual Stud...	3 KB
Project.VC	2/20/2017 3:43 PM	Data Base File	220 KB
ZERO_CHECK	2/20/2017 3:41 PM	VC++ Project	41 KB
ZERO_CHECK.vcxproj	2/20/2017 3:41 PM	VC++ Project Filters F...	1 KB

Figure 2: Compiled SBL Library for VS (molecular distances)

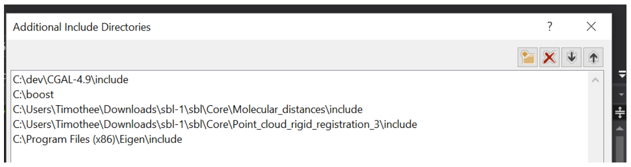
After all compilation done, we included the includes folder directly into the project :



Additional Library Directories

```
C:\dev\CGAL-4.9\build\lib
C:\dev\CGAL-4.9\build\lib\Debug
```

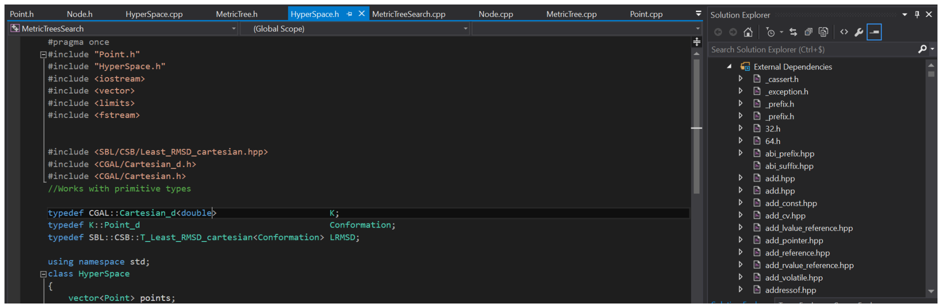
Figure 3: Include CGAL libraries



```
C:\dev\CGAL-4.9\include
C:\boost
C:\Users\Timothee\Downloads\sbl-1\sbl\Core\Molecular_distances\include
C:\Users\Timothee\Downloads\sbl-1\sbl\Core\Point_cloud_rigid_registration_3\include
C:\Program Files (x86)\Eigen\include
```

Figure 4: Include SBAL Libraries

Here is an example with the external dependencies used in the project :



```cpp
#pragma once
#include "Point.h"
#include "HyperSpace.h"
#include <iostream>
#include <vector>
#include <limits>
#include <fstream>


#include <SBL/CSB/Least_RMSD_cartesian.hpp>
#include <CGAL/Cartesian_d.h>
#include <CGAL/Cartesian.h>
//Works with primitive types

typedef CGAL::Cartesian_d<double>                     K;
typedef K::Point_d                                    Conformation;
typedef SBL::CSB::T_Least_RMSD_cartesian<Conformation> LRMSD;

using namespace std;
class HyperSpace
{
    vector<Point> points;
```

```
External Dependencies
    _cassert.h
    _exception.h
    _prefix.h
    _prefix.h
    32.h
    64.h
    abi_prefix.hpp
    abi_suffix.hpp
    add.hpp
    add.hpp
    add_const.hpp
    add_cv.hpp
    add_lvalue_reference.hpp
    add_pointer.hpp
    add_reference.hpp
    add_rvalue_reference.hpp
    add_volatile.hpp
    addressof.hpp
```

Figure 5: Included libraries in practice

3 Metric space

3.1 Reminders

A metric space is a set for which distances between all members of the set are defined. Those distances, taken together, are called a metric on the set.

Definition : A metric space is a pair (M,d), with $d : M \times M \to$ R, such that for any **x,y,z** $\in$ **M**, the following holds :

1. Positivity : d(x,y) $\geq$ 0
2. Self-distance : d(x,x) = 0
3. Isolation : x $\neq$ y $\Rightarrow$ d(x,y) $\geq$ 0
4. Symmetry : d(x,y) = d(y,x)
5. Triangle inequality : d(x,z) $\leq$ d(x,y) + d(y,z)

3.2 Implementation of the HyperSpace in C++

In order to create the HyperSpace, we've created two different constructors : a first one for the creation of an hyperspace for which the points have a uniform distribution and an other for which the points have a gaussian distribution. Here is the constructor for the HyperSpace with uniform distribution of points :

```
HyperSpace::HyperSpace(int pointsCount, int pointsDimension ,
int dimensionSize,int bound)
{
    this->pointsCount = pointsCount;
    this->dimensionSize = dimensionSize;
    this->bound = bound;

    for (int i = 0; i < pointsCount; i++) {
        Point p =  Point(dimensionSize, bound);
        if (pointsDimension < dimensionSize) {
            for (int j = pointsDimension; j < dimensionSize; j++) {
                p.setAt(j, 0.0f);
            }
        }
        this->points.push_back(p);
    }
}
```

As you can see, the constructor takes several arguments which are :

- pointsCount : total number of points in the HyperSpace
- pointsDimension : dimension of the points generated
- dimensionSize : total number of dimensions
- bound : is used to bound the value of the points generated

Concerning the constructor of the HyperSpace with a gaussian distribution, we added two arguments in addition to the previous ones :

- mean : Mean of the gaussian distribution
- stdDev : Standard deviation

The code used for this constructor is almost the same, we just added two parameters to the following line :

```
Point p = Point(dimensionSize, bound, mean, stdDev);
```

The function Point allow us to generate points in our space either with a uniform distribution or a gaussian distribution. By default this function will generate points with the same dimensions as the hyperspace. As we want to be able to have points with a lower number of dimensions than the HyperSpace, we created the for loop which allow us to ”remove” a given number of dimensions (depending on the parameter pointsDimension).

3.3 Implementation of distances in C++

3.3.1 Euclidian distance

We implemented two different distances in our HyperSpace : the Euclidian distance and the LRMSD distance.

The Euclidian distance between two points **a** and **b** is the length of the line segment connecting them. Here is the formula of the Euclidian distance in Cartesian coordinates between two points $\mathbf{a} = (a_1, a_2, ..., a_n)$ and $\mathbf{b} = (b_1, b_2, ..., b_n)$:

$$\mathrm{d}(\mathbf{a},\mathbf{b}) = \mathrm{d}(\mathbf{b},\mathbf{a}) = \sqrt{(b_1-a_1)^2 + (b_2-a_2)^2 + \cdots + (b_n-a_n)^2} \quad (1)$$

$$= \sqrt{\sum_{i=1}^{n}(b_i-a_i)^2}. \quad (2)$$

Here is the C++ code we used in order to implement this distance :

```
float HyperSpace::EuclidianDistance(Point & a, Point & b)
{

    int dimension = a.getDimension() == b.getDimension() ? a.getDimension↩
        () : 0;
    if (dimension == 0)
        return 0.0f;

    float dist = 0.0f;
    for (int k = 0; k < dimension; k++) {
        dist += (float)pow((float)a.getAt(k) - (float)b.getAt(k), 2.0f);
    }
    dist = (float)sqrt(dist);
    return dist;
}
```

First we make sure that the dimension of the first point is equal to the dimension of the second point. Then if it's the case, we implement the formula above.

3.3.2 lRMSD Distance

The least Root Mean Square Deviation is widely used in the study of molecules. The RMSD is the square root of the average of the squared distances between corresponding atoms of a pair of structures x and y. It is a measure of the average atomic displacement between two conformations. We can define the RMSD distance between two point sets A = $\{a_i\}_{i=1,......,n}$, B = $\{b_i\}_{i=1,.....,n}$ with the following formula :

$$RMSD(A,B) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} \|a_i - b_i\|_2^2} \tag{3}$$

By consequence, the least Root Mean Square Deviation can be defined with the following formula :

$$lRMSD(A,B) = \min_{g \in SE(3)} RMSD(A, g \cdot B) \tag{4}$$

In order to implement this distance, we used the module lRMSD distance of the SBL library as you can see below :

```
float HyperSpace::LRMSDDistance(Point & a, Point & b)
{
    vector<double> coords_p;
    vector<double> coords_q;

    for (int i = 0; i < a.getDimension(); i++) {
        coords_p.push_back(a.getAt(i));
    }
    for (int i = 0; i < b.getDimension(); i++) {
        coords_q.push_back(b.getAt(i));
    }
    Conformation p(coords_p.size(), coords_p.begin(), coords_p.end());
    Conformation q(coords_q.size(), coords_q.begin(), coords_q.end());


    LRMSD distance;
    double dist = distance(p, q);
    return dist;
}
```

First we transform both points into a conformation (as defined in the SBL library) and then we use the LRMSD distance in order to calculate the distance between this two conformations.

4 Metric trees

4.1 Reminders

A metric tree is any tree data structure specialized to index data in metric spaces. Metric trees exploit properties of metric spaces such as the triangle inequality to make accesses to the data mode efficient.

Definition : A metric tree is a binary tree for which any internal node implements a spherical cut defined by the distance μ to a pivot v :

- right subtree : points p such that d(pivot,p)$\geq \mu$
- left subtree : points p such that d(pivot,p) $\leq \mu$

4.2 Pseudo code

In order to build our metric tree, we chose to use a recursive construction. First we choose a pivot which ideally induce a partition into subset of the same size and then we assign points to the subtrees and we recurse.

```
**Data:** $S$ : Set with every points in the metric space
**Result:** Metric tree
{build_MetricTree($S$)};
if $S \neq \emptyset$ then
    return NIL;
end
n $\leftarrow$ $newNode$;
Draw at random $Q \subset S$ and $v \in Q$;
n.pivot $\leftarrow$v;
$\mu \leftarrow$ median({d(v,p),p $\in$ Q\{p}});
{The pivot splits points into two subsets};
$L \leftarrow \{s \in S \setminus \{p\} \mid d(s,v) \leq \mu\}$;
$R \leftarrow \{s \in S \setminus \{p\} \mid d(s,v) \geq \mu\}$;
{For each subtree: min / max distances to points in that subtree };
$n.(d_1,d_2) \leftarrow$ (min,max) of distances $d(v,p),p \in L$;
$n.(d_3,d_4) \leftarrow$ (min,max) of distances $d(v,p),p \in R$;
{Recursion};
$n.L \leftarrow$ build_MetricTree(L);
$n.R \leftarrow$ build_MetricTree(R);
```

Algorithm 1: Building a metric tree

Our code is based on this algorithm concerning the implementation of the metric trees.

4.3 Implementation in C++

In order to be able to cover every configurations, we created 4 different functions for the implementations of the metric trees.

4.3.1 Basic metric tree

Here is function we created in order to build a basic metric tree :

```
void MetricTree::buildMetricTreeBasic(vector<Point> listPoints,
Node * currentNode)
{
    this->allNodes.push_back(currentNode);
    if (listPoints.size() <= 1) {
        currentNode->setLeafTrue();
        if (listPoints.size() == 1) {
            currentNode->setPivot(listPoints.at(0));
        }
}
```

As you can see below, this is a recursive function which takes as parameters the set of points in the metric space and the current node on which the function is iterating.

First we check the stopping criteria. In our case, we check if the quantity of point left in the set of points is inferior or equal to 1. If it does, we set the node as being a leaf and we set the current node as a pivot.

```
int listPointsSize = listPoints.size();
int randNumberOfPoints = 1 + (rand() % (int)(listPointsSize));
vector<int> pointsSelected;
do {
    int randNum = (rand() % (int)(listPointsSize));
    if (!(find(pointsSelected.begin(), pointsSelected.end(),
        randNum) != pointsSelected.end())) {
        pointsSelected.push_back(randNum);
    }
} while (pointsSelected.size() < randNumberOfPoints);
```

First we start by choosing a random number of points among the existing ones and then we select randomly the amount of point in the data set. We store those values in the vector "pointsSelected".

```
Point randPivot = listPoints.at(pointsSelected.at(0));
currentNode->setPivot(randPivot);
map<Point*, float> m;
vector<float> allDistances;
for (int i = 0; i < randNumberOfPoints; i++) {
    float dist = this->hyperSpace->EuclidianDistance(
    randPivot, listPoints.at(pointsSelected.at(i)));
    m[&listPoints.at(pointsSelected.at(i))] = dist;
    allDistances.push_back(dist);
}
```

We affect arbitrary the first point of the vector as a pivot and it will be optimized later by the pivot selection. Then we compute all the distances between the pivot and the points we randomly chose previously (*randNumberOfPoints*).

```
float median = this->median(allDistances);
vector<Point> leftTree;
vector<Point> rightTree;
map<Point*, float> m2;
for (int i = 0; i < listPointsSize; i++) {
    float dist = this->hyperSpace->EuclidianDistance(
    randPivot, listPoints.at(i));
    m2[&listPoints.at(i)] = dist;
}
```

We compute the median of the distances and we store it into the *median* variable. We define two vectors of points, one for the left part of the tree and a second one for the right part. Then we compute the distance between the pivot and all the remaining points in the space. We store those values in the array *m2*.

```
map<Point*, float >::iterator it;
for (it = m2.begin(); it != m2.end(); it++)
{
    if (it->second<median) {
        leftTree.push_back(*it->first);
    }
    else {
        rightTree.push_back(*it->first);
    }
}
float d1 = numeric_limits<float >::max();
float d3 = numeric_limits<float >::max();
float d2 = numeric_limits<float >::min();
float d4 = numeric_limits<float >::min();
```

We iterate on each points of the space and we compare the value of their median to the median we previously computed. If it's inferior, the point goes to the left subtree and if it's superior, it goes to the right subtree. We initialize d1, d2, d3 and d4 which will be used to store the min / max distances to each points in the two different subtrees.

```
for (int i = 0; i < leftTree.size(); i++) {
    float distPivotPoint = this->hyperSpace->EuclidianDistance(randPivot, ↩
        leftTree.at(i));
    if (distPivotPoint <= d1) {
        d1 = distPivotPoint;
    }
    if (distPivotPoint >= d2) {
        d2 = distPivotPoint;
    }
}

for (int i = 0; i < rightTree.size(); i++) {
    float distPivotPoint = this->hyperSpace->EuclidianDistance(randPivot, ↩
        rightTree.at(i));
    if (distPivotPoint <= d3) {
        d3 = distPivotPoint;
    }
    if (distPivotPoint >= d4) {
        d4 = distPivotPoint;
    }
}
currentNode->setD(1, d1);
currentNode->setD(2, d2);
currentNode->setD(3, d3);
currentNode->setD(4, d4);

currentNode->left = new Node();
buildMetricTreeBasic(leftTree, currentNode->left);
currentNode->right = new Node();
buildMetricTreeBasic(rightTree, currentNode->right);
```

We create a loop for each subtree and we get the min and max distance for each ones. Then we update the currentNode values and we recall the function on each subtrees.

4.3.2 Basic metric tree on conformations

This function is also an implementation of a metric tree but instead of using the Euclidian distances as we did in the previous function, we use the lRMSD distance.

```
for (int i = 0; i < randNumberOfPoints; i++) {
    float dist = this->hyperSpace->LRMSDDistance(randPivot, listPoints.at(↩
        pointsSelected.at(i)));
    m[&listPoints.at(pointsSelected.at(i))] = dist;
    allDistances.push_back(dist);
}

float median = this->median(allDistances);
```

```
vector<Point> leftTree;
vector<Point> rightTree;

map<Point*, float> m2;
for (int i = 0; i < listPointsSize; i++) {
    float dist = this->hyperSpace->LRMSDDistance(randPivot, listPoints.at(↩
        i));
    m2[&listPoints.at(i)] = dist;
}
```

4.3.3 Optimized metric tree

The implementation of the optimized metric tree is almost the same as the basic metric tree. The only thing that changes is the way we select the pivot. Indeed, in the basic metric tree, we selected the pivot randomly and in this case, we use the function getBestPivot.

```
vector<Point> pointsSelectedValues;
for (int i = 0; i < pointsSelected.size(); i++) {
    pointsSelectedValues.push_back(listPoints.at(pointsSelected.at(i)));
}

Point optimizedPivot = this->getBestPivot(pointsSelectedValues);

currentNode->setPivot(optimizedPivot);
```

The function getBestPivot take as paramter a vector of points.

```
Point MetricTree::getBestPivot(vector<Point> listPoints)
{
    if(listPoints.size()==1){
        return listPoints.at(0);
}else if (listPoints.size() > 0) {
```

It check if the vector is not empty and then get the mean value of each points per dimensions.

```
int dim = listPoints.at(0).getDimension();
```

```
float * values = new float[dim];

for (int i = 0; i < dim; i++) {
    float sumDim = 0.0;
    for (int j = 0; j < listPoints.size(); j++) {
        sumDim += listPoints.at(j).getAt(i);
    }
    float meanDim = sumDim / listPoints.size();
    values[i] = meanDim;
}
```

Now we can create a point in the center of the vector of points. Then we compute the Euclidian distance between each points and the point at the center. The best pivot will be the point with the biggest Euclidiant distance from the point at the center.

```
Point center = Point(dim, values);
float maxDistSoFar = 0.0;
Point bestPivotSoFar = listPoints.at(0);

for (int j = 0; j < listPoints.size(); j++) {
    float currentDist = this->hyperSpace->EuclidianDistance(center, ↩
        listPoints.at(j));
    if (currentDist > maxDistSoFar) {
        maxDistSoFar = currentDist;
        bestPivotSoFar = listPoints.at(j);
    }
}

return bestPivotSoFar;
```

4.3.4 Optimized metric tree on conformations

The last function we implemented concerning the metric trees is the optimized metric tree on conformations. It works the same way as the previous function, but instead of using the Euclidian distances, it uses the lRMSD distances.

5 Search strategy

In order to be able to search for a specific point into a metric space, we use different kind of search strategy. Even if they are different, all of them are based on the use of a metric tree as a basis.

5.1 Pseudo code

```
Data: $T$ : Node of the metric tree, $q$ : point we're looking for
Result:
{Note of $T$ is denoted $n$};
$nn$(q) $\leftarrow \emptyset$;
$\tau \leftarrow \infty$;
if  $n = NIL$ then
    return
end
{build_MetricTree($S$)};
if $S \neq \emptyset$ then
    return NIL;
end
{Check whether the pivot is the nn};
$I \leftarrow$ d(q,n.pivot);
if $I \leq \tau$ then
    $nn$(q) $\leftarrow$ n.pivot;
    $\tau \leftarrow I$;
end
{Dilate the distance intervals for left and right subtree};
$I_l \leftarrow [n.d_1 - \tau, n.d_2 + \tau]$;
$I_r \leftarrow [n.d_3 - \tau, n.d_4 + \tau]$;
{Recursion};
if $I \in I_l$ then
    search_MetricTree(n.$L$,q);
end
if $I \in I_r$ then
    search_MetricTree(n.$R$,q);
end
```

Algorithm 2: Searching a metric tree

5.2 Defeatist strategy

The defeatist strategy will search a point through a metric tree and will check every nodes until he finds the point. When he does, the algorithms stops by himself. We did two implementation of this algorithm in C++, one for the Euclidian distance and an other one for the lRMSD distance (*search-MetricTreeDefeatistConformation*). We will present you our implementation in C++ with the Euclidian distance :

```
bool MetricTree::searchMetricTreeDefeatist(Node * T, Point * q)
{
    float tau = 0.0;
    if (this->numberOfNodeExplored == 0.0) {
        tau = std::numeric_limits<float >::max();
    }
    Point pivot = T->getPivot();
    if (pivot == *q) {
        return true;
    }
    if (T->isALeaf()) {
        return false;
    }
```

First we intialize tau and we check if the number of node explored is equal to zero. If it is, we set tau to the max of the numeric values in C++. Then we check if the pivot of the node is the point we're looking for. If it is, we stop the search here. If not, we continue and we check if the node is a leaf.

```
float I = this->hyperSpace->EuclidianDistance(pivot, *q);

if (I < tau) {
    tau = I;
    this->nearestNeighbour = pivot;
}

float Ilmin = T->getD(1) - tau;
float Ilmax = T->getD(2) + tau;
float Irmin = T->getD(3) - tau;
float Irmax = T->getD(4) + tau;

if ((I >= Ilmin) & (I <= Ilmax) & !this->found) {
        this->numberOfNodeExplored++;
        this->searchMetricTreeDefeatist(T->left, q);
    }
    else{
        this->numberOfNodeExplored++;
        this->searchMetricTreeDefeatist(T->right, q);
}
```

Then we compute the Euclidian distance between the pivot and the point we're looking for. If that distance is smaller than tau, we define our nearest neighbor as being the pivot. Then we delimit the new area we have to look into and we do a recursive call on one of the subtree depending on the distance between the point and the pivot.

5.3 Exact search strategy

The exact search strategy is almost the same as the defeatist one but before the recursive call, we add a pruning condition, as you can see below :

```
if ((I >= Ilmin) & (I <= Ilmax)) {
    this->numberOfNodeExplored++;
    this->searchMetricTreePrunning(T->left, q);
}
if ((I >= Irmin) & (I <= Irmax)) {
    this->numberOfNodeExplored++;
    this->searchMetricTreePrunning(T->right, q);
}
```

We implemented two function for this algorithm, one with the Euclidian distance and an other one with the lRMSD distance.

6 Execution full example on command

Once the program was done, we created an executable in order to be able to launch it through a command line interface. Here is an example of it's execution.

Listing 1: Parameters inputs

```
Type the dimension of the hyperspace : Number of points - Points Dimension↩
    - Dimension of the space - Max absolute value - Optional Mean - ↩
    Optional Standard Deviation
type parameters (4 or 6 separated by space) : 3 2 2 100 9 60

Hyper space
0: -9.1908, -58.6441,

1: -1.06371, 31.9912,
2: 9.80256, 83.1871,

What metric tree do you want to build ?
1: Basic
2: Basic on conformations
3: Optimized
4: Optimized on conformations
3
Type 1 if you want to print the metric tree
1
```

First the program ask for a small set of parameters which are the number of points, the dimensions of the points and of the metric space, the bound, the mean and the standard deviation (only if we want a gaussian distribution). If we don't precise the two last parameters, the generation of the points will be following a uniform distribution. Then the program generated the points requested and ask for the metric tree we want.

Listing 2: Metric tree output

```
Begin METRIC TREE ----------------------
Racine : Begin Node 10437344----------------------
isLeaf 0
Pivot : -1.06371, 31.9912,

d1 : 3.40282e+38 d2 : 1.17549e-38 d3 : 0 d4 : 90.9989
End Node ----------------------

Node or leaf count number : 0
Begin Node 10437344----------------------
isLeaf 0
Pivot : -1.06371, 31.9912,

d1 : 3.40282e+38 d2 : 1.17549e-38 d3 : 0 d4 : 90.9989
End Node ----------------------

LEFT NODE : 10484952
RIGHT NODE : 10485048
Node or leaf count number : 1
Begin Node 10484952----------------------
isLeaf 1
Pivot :

d1 : 0 d2 : 0 d3 : 0 d4 : 0
End Node ----------------------

.....

nombre de node 5
nombre de feuille 6
End METRIC TREE ----------------------
```

the program build the metric tree we asked for. As you can see, each node has a pivot, ids of left and right node and Ds values

Listing 3: Searching a known point in the metric tree

```
Lets search point, the number of point in the hyperspace is : 3
Type a number between 0 and 2 included to search the point at this ↩
    position
Otherwise type 99999 if you want to search a point that doesn t exist to ↩
    test it !
1
What type of search do you want to do ?
1: Prunning condition
2: Defeastist
3: Both
3
Defeatist
Node explored °n : 0
search Metric function called
Current Node explored : Begin Node 10437344−−−−−−−−−−−−−−−−−−−−−−−
isLeaf 0
Pivot : −1.06371, 31.9912,

d1 : 3.40282e+38 d2 : 1.17549e−38 d3 : 0 d4 : 90.9989
End Node −−−−−−−−−−−−−−−−−−−−−−−

Point has been found at this Node
we searched :−1.06371, 31.9912,

Number of node explored : 0
Prunning
Node explored °n : 0
search Metric function called
Current Node explored : Begin Node 10437344−−−−−−−−−−−−−−−−−−−−−−−
isLeaf 0
Pivot : −1.06371, 31.9912,

d1 : 3.40282e+38 d2 : 1.17549e−38 d3 : 0 d4 : 90.9989
End Node −−−−−−−−−−−−−−−−−−−−−−−

Point has been found at this Node
we searched :−1.06371, 31.9912,
```

One the metric tree is generated, the program as you which point to search and which search strategy you want to use. Then for each node of the metric tree that the search strategy have to visit, you have a few informations such as the pivot location. The algorithm stops when it finds the point.

Listing 4: Searching a point that does not exist in the hyperspace Defeatist strategy

```
Lets search point , the number of point in the hyperspace is : 3
Type a number between 0 and 2 included to search the point at this ↩
    position
Otherwise type 99999 if you want to search a point that doesn t exist to ↩
    test it !
99999
What type of search do you want to do ?
1: Prunning condition
2: Defeastist
3: Both
3
Defeatist
Node explored °n : 0
search Metric function called
Current Node explored : Begin Node 10437344−−−−−−−−−−−−−−−−−−−−−−−
isLeaf 0
Pivot : −1.06371, 31.9912,

d1 : 3.40282e+38 d2 : 1.17549e−38 d3 : 0 d4 : 90.9989
End Node −−−−−−−−−−−−−−−−−−−−−−−


...............


I : 129.126
tau : 0
Ilmin : 0
Ilmax : 1.17549e−38
Irmin : 52.3363
Irmax : 52.3363

Node explored °n : 5
search Metric function called
Current Node explored : Begin Node 10504848−−−−−−−−−−−−−−−−−−−−−−−
isLeaf 1
Pivot : 9.80256, 83.1871,

d1 : 0 d2 : 0 d3 : 0 d4 : 0
End Node −−−−−−−−−−−−−−−−−−−−−−−

we searched :76.513, −71.2333,

Number of node explored : 5
The point hasn t been found !
```

You can also ask to the program to search a point that is not existing ! In this case, you can see the way the algorithm behave when it can't find the point. Here it is the result for the defeatist strategy.

Listing 5: Searching a point that does not exist in the hyperspace Prunning strategy

```
Prunning
Node explored °n : 0
search Metric function called
Current Node explored : Begin Node 10437344----------------------
isLeaf 0
Pivot : −1.06371, 31.9912,

d1 : 3.40282e+38 d2 : 1.17549e−38 d3 : 0 d4 : 90.9989
End Node ----------------------

I : 129.126
tau : 129.126
Ilmin : 3.40282e+38
Ilmax : 129.126
Irmin : −129.126
Irmax : 220.125

Node explored °n : 1
search Metric function called
Current Node explored : Begin Node 10485048----------------------
isLeaf 0
Pivot : −9.1908, −58.6441,

d1 : 0 d2 : 1.17549e−38 d3 : 90.9989 d4 : 143.097
End Node ----------------------

I : 86.6235
tau : 0
Ilmin : 0
Ilmax : 1.17549e−38
Irmin : 90.9989
Irmax : 143.097

we searched :76.513, −71.2333,

Number of node explored : 1
The point hasn t been found !
```

And here it is the same configuration but for the pruning conditions. We can notice that the prunning condition stops the search very quickly compared to the defeatist strategy.

7 Study of the results

We've run this program in different configuration and with different parameters in order to analyses the performances of each metric tree and search strategies. We used the number of node visited before the search strategies found (or not) the requested point. Here are our results :

7.1 Simple uniform data generation

In this configuration, the point of the metric space which are generated are following a uniform law of distribution. Each time we run the program, we have to input several parameters which are :

- Number of points : number of points we want to generate in the metric space
- Dimension of the points : dimension of the points which can't be higher than the dimension of the metric space.
- Dimension of the metric space
- Absolute bounding : bounds of the metric space

In the following tables, the parameters are exactly in the same order as presented previously.

7.1.1 Basic metric tree

Each time we generate a new dataset (with different parameters), we run both the search strategies on 4 points each time and we analyze the results.

Parameters	Found	Nodes explored DS	Found	Nodes explored PC
1000, 5, 5, 100	no	10	yes	13
	yes	8	yes	8
	no	15	yes	14
	yes	13	yes	13
10000, 5, 5, 100	no	15	yes	11
	yes	15	yes	15
	no	12	yes	24
	yes	11	yes	11
10000, 8, 10, 100	no	16	yes	21
	yes	11	yes	11
	no	14	yes	22
	no	19	yes	29
15000, 12, 12, 100	yes	9	yes	9
	no	18	yes	26
	no	17	yes	19
	yes	14	yes	14

Table 1: Table of node exploration for a metric tree in uniform metric space

As we can see below and as expected, the search strategy with pruning conditions (PC) does always find the point we're looking for and the defeatist search doesn't always find it. We can also observe that each time the defeatist search finds the point we're looking for, the search strategy with pruning conditions finds it with the same amount of node visited.

7.1.2 Optimized metric tree

We run the same set of parameters as previously but with an optimized tree this time.

Parameters	Found	Nodes explored DS	Found	Nodes explored PC
1000, 5, 5, 100	yes	5	yes	5
	no	15	yes	12
	no	15	yes	15
	yes	9	yes	9
10000, 5, 5, 100	no	18	yes	14
	yes	18	yes	18
	no	17	yes	16
	yes	11	yes	15
10000, 8, 10, 100	no	13	yes	9
	yes	14	yes	14
	yes	11	yes	11
	no	18	yes	20
15000, 12, 12, 100	yes	11	yes	11
	yes	9	yes	9
	no	23	yes	19
	yes	15	yes	15

Table 2: Table of node exploration for an optimized metric tree in uniform metric space

We can do the same observation as previously concerning the performance of the two different search strategies.

7.1.3 Basic metric tree on molecular conformations

As explained previously, the basic metric tree on molecular conformations is using the lRMSD distance. We did run a different set of parameters as the libraries we used did require high bounds and dimensions. As our computer are not very powerful, we could only generate a few number of points.

Parameters	Found	Nodes explored DS	Found	Nodes explored PC
300, 333, 333, 99999	no	10	yes	16
	no	10	yes	16
	yes	11	yes	11
	yes	13	yes	13
	no	15	yes	15
300, 30, 30, 99999	yes	10	yes	10
	no	13	yes	16
	no	10	yes	11
	yes	8	yes	8
	yes	8	yes	8
600, 30, 30, 99999	yes	8	yes	8
	yes	10	yes	10
	yes	10	yes	10
	yes	6	yes	6
	no	20	yes	18
600, 150, 150, 999999	no	13	yes	19
	yes	4	yes	4
	no	13	yes	14
	yes	8	yes	8
	no	13	yes	19

Table 3: Table of node exploration for a basic metric tree for molecular conformations in a uniform metric space

Our observations are pretty much the same as before, except that the defeatist search did perform better two times.

7.1.4 Optimized metric tree on molecular conformations

Here are our results for the optimized metric tree on molecular conformations.

Parameters	Found	Nodes explored DS	Found	Nodes explored PC
300, 333, 333, 99999	yes	0	yes	0
	no	10	yes	17
	yes	6	yes	6
	no	15	yes	13
	no	8	yes	8
300, 30, 30, 99999	no	11	yes	11
	yes	6	yes	6
	yes	2	yes	2
	yes	9	yes	9
	no	12	yes	16
600, 30, 30, 99999	yes	6	yes	6
	no	14	yes	16
	no	11	yes	17
	no	10	yes	17
	yes	12	yes	12
600, 150, 150, 99999	yes	8	yes	8
	no	13	yes	11
	yes	8	yes	8
	no	10	yes	7
	yes	6	yes	11
	no	11	yes	16

Table 4: Table of node exploration for an optimized metric tree for molecular conformations in a uniform metric space

7.2 Simple gaussian data generation

The second part of our results are done on a metric space with points which are generated following a gaussian distribution. This is why we have to more parameters in our table, which are :

- mean : the mean of the distribution
- stdrDeviation : standard deviation of the distribution

7.2.1 Basic metric tree

Parameters	Found	Nodes explored DS	Found	Nodes explored PC
100, 3, 3, 100, 50, 20	no	8	yes	5
	yes	6	yes	6
	no	8	yes	8
	no	13	yes	19
100, 15, 15, 100, 50, 20	yes	6	yes	6
	yes	5	yes	5
	yes	5	yes	5
	yes	6	yes	6
100, 5, 5, 1000, 500, 200	no	10	yes	8
	no	8	yes	8
	no	12	yes	16
	yes	10	yes	10
100, 15, 15, 1000, 500, 200	yes	4	yes	4
	yes	11	yes	11
	no	7	yes	8
	yes	6	yes	6

Table 5: Table of node exploration for a metric tree in gaussian metric space

Our observations are pretty much the same as before.

7.2.2 Optimized metric tree

Parameters	Found	Nodes explored DS	Found	Nodes explored PC
100, 3, 3, 100, 50, 20	no	8	yes	9
	yes	5	yes	5
	no	7	yes	7
	no	11	yes	11
100, 15, 15, 100, 50, 20	no	8	yes	4
	no	7	yes	7
	no	8	yes	7
	yes	5	yes	5
100, 5, 5, 1000, 500, 200	no	6	yes	6
	yes	6	yes	6
	no	6	yes	5
	no	10	yes	7
100, 15, 15, 1000, 500, 200	yes	8	yes	8
	yes	5	yes	5
	no	10	yes	11
	no	10	yes	8

Table 6: Table of node exploration for an optimized metric tree in gaussian metric space

7.2.3 Basic metric tree on molecular conformations

Parameters	Found	Nodes explored DS	Found	Nodes explored PC
	no	12	yes	8
	no	9	yes	7
100, 30, 30, 99999, 50000, 20000	yes	8	yes	8
	no	17	yes	15
	no	8	yes	13
	no	9	yes	11
	no	12	yes	11
100, 15, 15, 99999, 50000, 20000	yes	7	yes	7
	yes	8	yes	8
	no	4	yes	4
	no	7	yes	4
	no	7	yes	11
100, 15, 15, 99999, 10000, 5000	no	7	yes	7
	yes	5	yes	5
	no	7	yes	11
	yes	7	yes	7
	no	8	yes	3
100, 15, 15, 99999, 10000, 5000	no	9	yes	10
	yes	4	yes	4
	yes	9	yes	9

Table 7: Table of node exploration for an optimized metric tree in gaussian metric space

7.2.4 Optimized metric tree on molecular conformations

Parameters	Found	Nodes explored DS	Found	Nodes explored PC
	yes	11	yes	11
	yes	6	yes	6
100, 30, 30, 99999, 50000, 20000	yes	6	yes	6
	yes	6	yes	6
	yes	6	yes	6
	no	6	yes	6
	no	6	yes	6
100, 15, 15, 99999, 50000, 20000	no	6	yes	12
	no	6	yes	8
	no	6	yes	5
	yes	7	yes	7
	no	12	yes	7
100, 15, 15, 99999, 10000, 5000	yes	4	yes	4
	yes	11	yes	11
	no	6	yes	9
	yes	3	yes	3
	no	12	yes	10
100, 15, 15, 99999, 1000, 500	no	12	yes	4
	yes	9	yes	9
	yes	7	yes	7

Table 8: Table of node exploration for an optimized metric tree in gaussian metric space

8 Conclusion

The goal of our project was to create a program which is able to create a metric space with points generated with a uniform or a Gaussian distribution, different kind of metric trees and search strategies, in order to compare the performance of those. We could observe the fact that the search strategy with pruning conditions is most of the time the best option, in every possible configurations, as it always finds the results and is as efficient as the defeatist search when this last one is able to find the result.

This project was also the occasion to see the limit of our computers. Indeed, it was not possible for us to work with big enough dataset as we would have like to do. This is why our results are a little bit light and could be improved. An improvement could be to implement a C++ function able to read a text file containing a set of parameters and a second function able to output the results for every metric tree and both the search strategy, in a xml file. It would give a better insight on the behavior of the different metric tree and search strategy, on would allow us to improve them.

9 References

- http://www-sop.inria.fr/teams/abs/classes/centrale-FGMDA/course-material/fcazals-class-one.pdf
- https://en.wikipedia.org/wiki/Metric_tree
- http://sbl.inria.fr/doc/
- https://www.quizover.com/course/section/rmsd-and-lrmsd-molecular-distance-measures-by-openstax
- Article : Optimal Pivot Selection Method Based on the Partition and the Prunning Effect for Metric Space Indexes
- Paper : Special Section on Data Engineering, Hisashi KURASAWA, Daiji FUKUGAWA, Astuhiro TAKASU, Jun ADACHI