Bachelors thesis

Speeding up the computation of parallel curve segments in DTI data
Stage 2

Author:
Klaas Mussche

Supervisor:
Dr. H. Bekker

Abstract:
Diffusion Tensor Imaging can give insight in the brain's internal structure. Multi-Scale Fiber Tract Bundling is a recently developed technique used to visualize Diffusion Tensor Imaging data. This technique requires some expensive computations. It was believed that a grid search method could do some of this computations faster, but it turned out that this was not the case. However, a fast implementation is possible which can do some of these computations faster then naive implementations can do.

August, 2010

Contents

Chapter 1

Introduction

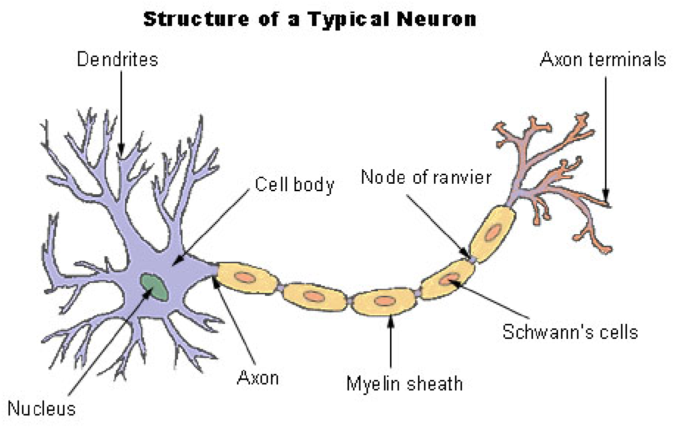


Figure 1.1: A typical neuron

The smallest building blocks of our brain are the neurons. A typical neuron is depicted in figure 1.1. It shows a nucleus on the left and an axon leaving the neuron. Neurons communicate with each other by using the axon terminals of the sending neuron and the dendrites of the receiving one.

The axons are surrounded with a white-colored matter called myelin. Because of this, people often refer to the brain as a combination of white and gray matter. Here the white matter refers to the structure used for the communication throughout the brain, while the gray matter is made up of the neurons doing the actual computations.

It is estimated in [6] that the human brain contain 50-100 billion (10^{11}) neurons, of which about 10 billion (10^{10}) pass signals to each other using as many as 1000 trillion (10^{15}) synaptic connections. Many axons are used for communication between different parts of the brain, for example between the left and right half. Because of this the axons are often bundled. Bundles of axons are sometimes called fibers.

Neuroscientists discovered that a damaged brain fiber structure probably causes conditions like schizophrenia and autism[5][4]. But understanding the brain fiber structure can also be useful for understanding learning processes[7].

To understand the brain structure, it is necessary that we can measure and visualize the fiber structure. In chapter 2 we will have a closer look at the *Diffusion Tensor Imaging* (or *DTI*) technique, which can be used for measuring the fiber structure.

These measurements however aren't ready for direct interpretation. Some kind of filtering and visualization is required in order to be able to understand and see the fiber structure of the brain. At the University of Groningen a technique to do this - called Multi-Scale Fiber Tract Bundling - has been developed[3]. In chapter 3 we will have a closer look at this technique.

This technique requires some time-consuming computations. The goal of my bachelor project was to find out if it is possible to speed up some of these computations. In a separate bachelor project Jorn van de Beek did similar research, however we did look at different ways to increase the performance.

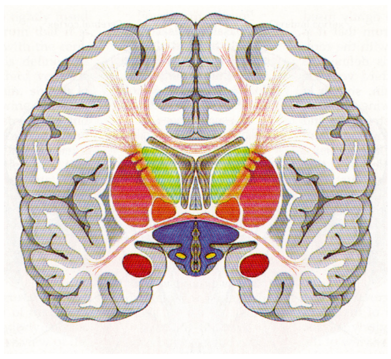


Figure 1.2: The human brain

Chapter 2

Diffusion Tensor Imaging

2.1 Magnetic Resonance Imaging

Magnetic Resonance Imaging (or MRI) is a technique used to capture 3D images of living tissues. Different types of living tissue will have a different magnetic resonance. It is possible, based upon the differences in magnetic resonance, to make a 3-D image of a body part. This image will contain the amount of magnetic resonance for every voxel. Using a filtering technique it is possible to get only the voxels with a magnetic resonance corresponding to the type of living tissue you are interested in.

Figure 2.1: MRI Scanner

MRI is widely used, both in clinics and research, and makes it possible to image body parts like spines, joints, abdomen, pelvis, blood vessels and the brain. While MRI is an interesting and useful technique, it isn't really suitable for measuring and visualizing the fiber structure of our brains. However, there is a variation on MRI, called Diffusion Tensor Imaging, which can be used to obtain information about the structure of our brains.

2.2 Diffusion Tensor Imaging

Axons are used for one-way communication between different neurons. They are bundled in nerve fibers, and the water in this fibers will diffuse more rapidly in the longitudinal direction of the fiber. In most tissues, like grey matter, the water molecules diffuse equally in all directions. This is called isotropic diffusion. But in nerve fibers, the diffusion is preferentially along the axis of the nerve. This is called anisotropic diffusion[1].

Using magnetic fields, it is possible to measure for every voxel the anisotropic diffusion. By directing a magnetic field through the body, it is possible to measure signal decay caused by diffusion.

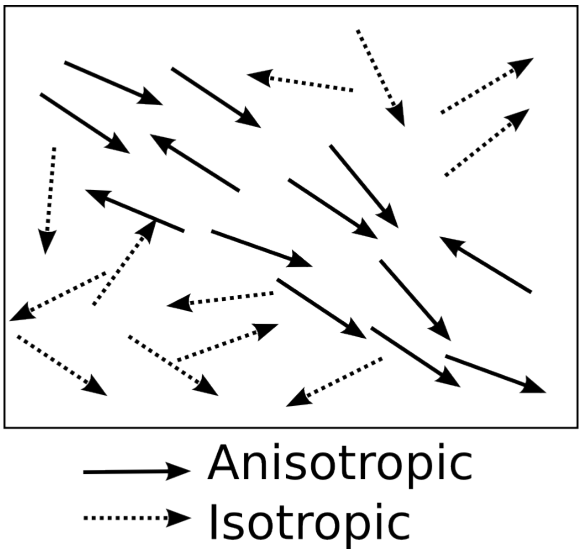


Figure 2.2: Isotropic and anisotropic diffusion

By measuring the degree of anisotropic diffusion in every voxel for the three principal directional axes, it is possible to calculate the dominant direction of anisotropic diffusion as a tensor. However, this doesn't contain enough information for DTI. For DTI it is necessary to know both the direction and the magnitude of the anisotropy in each voxel. This information can be used in two different ways. The first way is to make a cross section of the data and assign a color to every voxel based upon the relative amount of anisotropy.

The second way is to use mathematical algorithms to trace paths through the tensor field. For every path, a 3-D curve can be created and stored as a sequence of points. This is called tractography. If we want to reconstruct the structure of nerve fibers using tractography, then seven different measurements are needed[1]: six for measuring the actual diffusion, and one baseline measurement. Because a vector can't be used to store the information necessary for tractography, a mathematical concept known as a tensor is used to describe the anisotropy in each voxel.

In the context of diffusion tensor imaging, a scalar number can be seen as a tensor of rank 0, and a vector as a tensor of rank 1. A tensor of rank 2 can be described as a 3×3 matrix and each tensor of rank 2 can be described by nine measurements. Due to symmetry we only needed seven measurements for the tractography, so a tensor of rank 2 is enough.

2.3 Tractography

Tractography is a method used for reconstructing the trajectories of the fibers using the tensor field as measured by the *DTI* scanner. A possible tractography algorithm is described in [2].

One of the simplest possible tractography methods has been called Principal Diffusion Direction. This method starts tracking from user defined seed voxels and it follows the principal eigenvalue direction to trace a path until some stopping criterions are met. There are other comparable algorithms, but most of these algorithms result in errors due to discretization[2].

Tractography gives a set of tracts, where each tract is represented as a sequence of points (also called vertices) in space. The tract can be reconstructed by connecting this points. With standard tractography, the distance between each pair of consecutive points on the same tract might differ. On long straight tracts or subtracts it is possible to represent this with only two vertices. However, the algorithms following in this thesis assume that the tracts are equidistant sampled: the distance between every pair of consecutive vertices is constant. If this isn't the case, then the tracts have to be resampled before processing them.

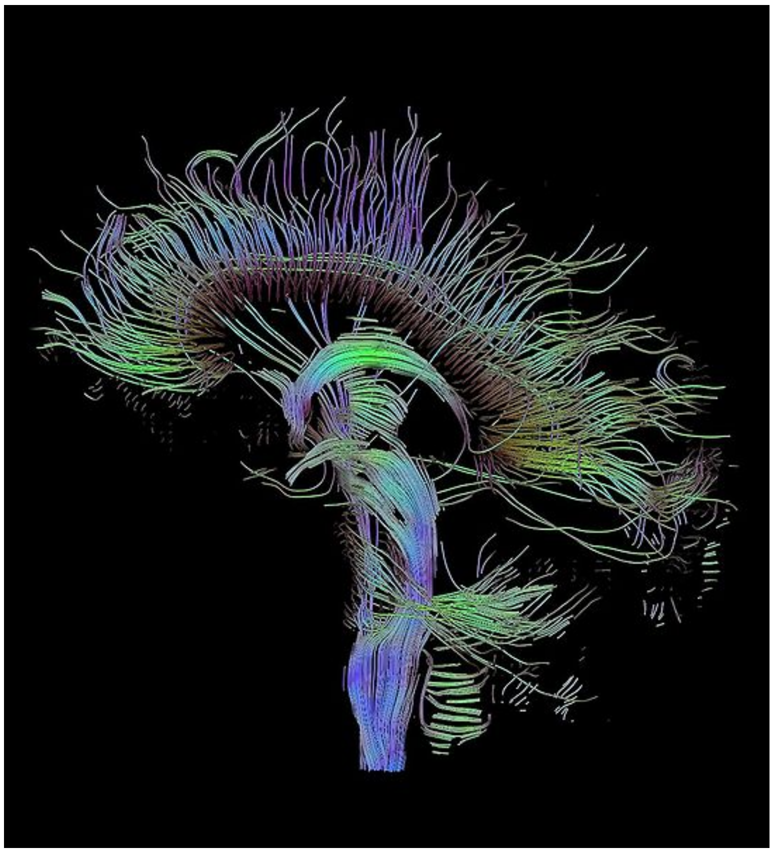


Figure 2.3: Results of tractography

Chapter 3

Multi-Scale Fiber Tract Bundling

3.1 Introduction

The Multi-Scale Fiber Tract Bundling technique as described in [3] is a visualization technique for brain fiber tracts from DTI data that provides insight into the structure of white matter. This technique analyzes the direction of all tracts and searches for tract segments close to each other with a similar direction. Those locally similar tract segments are bundled together. This leads to an abstraction of the global white matter structure and creates voids in between the data. Those voids decrease the mutual occlusion of tracts. The goal of this technique is to visualize entire DTI fiber tracts so that the depictions assist viewers in understanding the structure of white matter.

This technique starts by subdividing the tracts into small equal-sized subsegments. The process of subdividing the tracts into equal-sized subsegments is called resampling. Now each vertex will have one or two incident segments and for every pair of vertices it is possible to compute the similarity by looking at the directions of the corresponding incident segments. This computation will only be done when the distance between both vertices is below a certain treshold. When the similarity is above a certain treshold, then the vertices are considered locally similar.

The local similarity information is stored in a similarity graph. This graph is constructed from the resampled DTI data as follows. Every vertex corresponds to one node in the graph, and the graph contains an edge if and only if the vertices corresponding to the edges begin- and end node are locally similar. Based on this graph the tracts are iteratively bundled by relocating the tract vertices closer to similar ones.

Similar vertices will move closer to each other in the iterative bundling process and this will have the effect that similar tract segments are bundled.

When visualizing, the bundled tract data will have some advantages over the unbundled data. First, fiber bundles in the brains can be visualized by highlighting the tract bundles. Second, due to the bundling process, empty space is created between the bundles of data. These voids make it possible to see deeper into the brain.

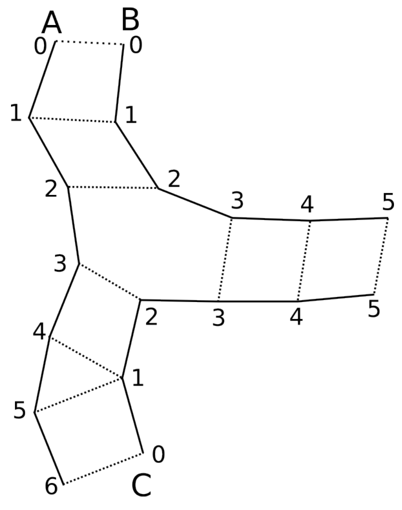


Figure 3.1: Nearest neighbor pairs

3.2 Analysis

The first part of the Multi-Scale Fiber Tract Bundling technique is the analysis of the tracts and construction of the similarity graph. This first part is called the analysis stage. A tract A with length $n-1$ consists of n vertices named $a_0, a_1, \dots, a_{n-1}$. Now this tract will have $n-1$ segments, $\vec{a}_0, \vec{a}_1, \dots, \vec{a}_{n-2}$. Each $\vec{a}_i$ is the segment connecting the vertices a_i and a_{i+1}. Each vertex a_i with $i > 0$ and $i < n-1$ does have two incident segments $\vec{a}_{i-1}$ and $\vec{a}_i$. If $i = 0$ or $i = n-1$ then there is just one incident segment.

Given two tracts A and B and a vertex p on A, the nearest neighbor of p on B, denoted $nn(p)$, is the vertex q on B so that the distance between p and q (denoted $D(p,q)$) is minimal, i.e. there is no vertex q' on B such that $D(p,q') < D(p,q)$. Now it is possible to construct a list L which contains all pairs (p,q) with p on A and q on B so that $p = nn(q)$ or $q = nn(p)$ and $(q,p) \notin L$. The last case is necessary to filter out so called superfluous entries, because the similarity graph is an undirected graph.

For every pair $(p,q) \in L$, an edge is created in the similarity graph between the nodes N_p and N_q if and only if the pair (p,q) satisfies the following requirements:

1. the distance between p and q is smaller than a given length d_{max}.
2. at least one of the segments incident to p is approximately parallel to one of the segments incident to q, where "approximately parallel" means that the angle between the segments is smaller than a predefined angle θ_{par}.
3. the nearest neighbor relation of p and q is approximately mutual, that is, it holds that $|index(nn(q)) - index(p)| \leq 1$ and $|index(nn(p)) - index(q)| \leq 1$, where the function $index(v)$ returns the index of v on the tract to which it belongs.

Here, the d_{max} is the threshold value for the distance between vertices and $\theta_{par}$is the threshold value used to determine if the vertex pair is similar. The third condition is used to prevent unwanted edge bundling during the bundling stage.

3.3 Bundling

The first stage of Multi-Scale Fiber Tract Bundling produces a similarity graph for a given DTI data set, but doesn't modify the data set and visualizing still gives the same result. The second stage modifies the actual data such that

- Bundles in the white matter structure are represented as bundles of tracts.
- Voids are created in the data set.

The first property will give insight in the *structure* of the white matter by abstracting the actual measured tracts to bundles of tracts. The voids are useful too, because they will reduce mutual occlusion during visualization.

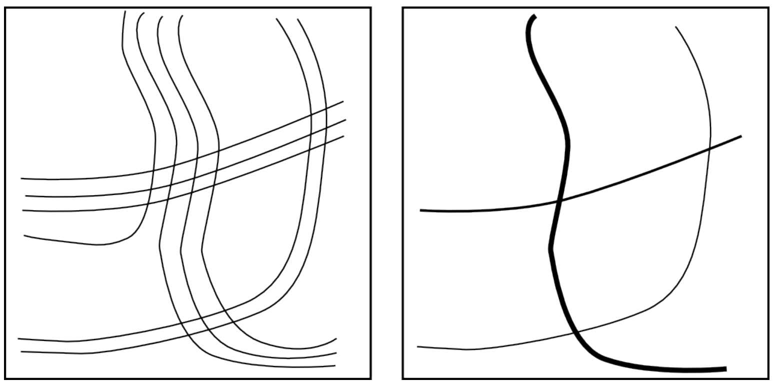


Figure 3.2: The concept of bundling

A simple bundling algorithm could use the information stored in the similarity graph to move locally similar vertices closer to each other. However, iterating over all edges in the similarity graph and displacing the vertices corresponding to each edge once isn't a good option because most vertices will have more then one similar vertex and so the ordering of the edges will influence the results of the bundling stage.

Let us consider two locally similar vertices p and q. If p and q don't have any other locally similar vertices, then the bundling process could move both p and q to the point halfway between the original locations of p and q. In most situations there will be other locally similar vertices and these will influence the new positions of either p or q.

The bundling stage computes the displacement vectors of all vertices before applying them. This ensures that the ordering of the edges doesn't have any influence on the results. To avoid run-away situations, each displacement vector is divided by the number of edges connected to the vertex. Also, the application of a small gaussian kernel to each displacement vector will prevent sudden transitions where merging tracts come into the influence range of other tracts.

Moving the vertices will alter the data. Ensuring that vertices will only be moved perpendicularly to the tracts local orientation will keep the amount of deformation to a minimum. The local orientation of a vertex can be computed as the average direction of both segments incident to the processed vertex.

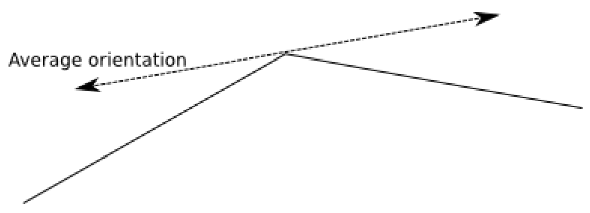


Figure 3.3: Local orientation

The choice of d_{max} influences the results of the bundling stage. To make it possible to explore the scale, a fairly large value d_{max} is used during the analysis stage and then the bundling stage is done for different increasing values $d_{max,i} \in [0, d_{max}]$.

3.4 Visualization

The bundling stage results in a set of tracts where not only the original position is stored for each vertex, but also the positions for the different values of $d_{max,i}$. Using this values, it is possible to visualize the data where the user can interactively and seamlessly move from one bundling scale to the next.

When rendering the tracts, the positions for vertices are computed using the stored positions for the involved vertex and the selected bundling scale. If the bundling scale equals one value $d_{max,i}$ then the position is stored directly. Otherwise, the position can be calculated by interpolating the positions corresponding to the two values of $d_{max,i}$ closest to the bundling scale. This interpolation can be done using a vertex shader. This makes it possible to render the tracts in realtime and thus enables the user to explore the white matter structure by continuously changing the rendering parameters. For example, users can play with the bundling scale. The continuous transition of the vertex position enables the user to follow the tracts and helps them to get insight in the white matter structure.

Visualization can be improved by doing some further filtering. The bundling of tracts results in voids, which decrease mutual occlusion. By filtering on thin structures, it is possible to further decrease the mutual occlusion. This filtering can be implemented by looking at the degree of each vertex in the similarity graph and omitting the vertices with a degree below a certain threshold.

Chapter 4

Naive approach

4.1 Introduction

The first possible analysis stage implementation of the Multi-Scale Fiber Tract Bundling technique is by just checking everything and selecting the vertex pairs satisfying all three conditions. This naive approach was used in [3] but it took too much time. However, we will have a look at this naive solution, because it can be useful for us when constructing a better algorithm.

The analysis stage has to find, given a set of tracts where each tract is an equidistant sampled set of vertices, all pairs of vertices (p, q) with p and q on different tracts which satisfy the three conditions for similar vertices described in section 3.2. The naive algorithm does this by iterating over all possible pairs of vertices (p, q) with p and q on different tracts. For every pair, the three conditions are checked and if all three conditions are satisfied, then the pair is added to the results.

Iteration over all nearest neighbor pairs (p, q) can be done with the following algorithm:

```
for $a = 0$ to $num\_tracts - 1$ do
  for $i = 0$ to $tract\_length[a] - 1$ do
    $p \Leftarrow tracts[a][i]$ {$p$ is on tract $a$ and has vertex index $i$}
    for $b = a + 1$ to $num\_tracts - 1$ do
      $md \Leftarrow \infty$
      for $j = 0$ to $tract\_length[b] - 1$ do
        $t \Leftarrow tracts[b][j]$ {$t$ is on tract $b$ and has vertex index $j$}
        if $D(p, t) \leq md$ then
          $md \Leftarrow D(p, t)$
          $q \Leftarrow t$
        end if
      end for
      if check$(p, q)$ then
```

```
            {The pair $(p, q)$ satisfies all three conditions.}
          end if
        end for
      end for
    end for
```

Here, the *check* function checks the three conditions for the given pair (p, q) and returns true if and only if all three conditions are satisfied.

4.2 Distance condition

The *check* method should compute the distance and compare this with the threshold d_{max}. Computing the Euclidean distance usually involves a square root computation. But square root computations are expensive and it is better to avoid them. In this case, it is possible to avoid square root computations by working with the squared distances and d_{max}^2, because comparisons between squared numbers give the same results as comparison between the normal numbers.

4.3 Angle condition

The angle condition can be verified by computing the (up to four) angles between pairs of incident segments. If one of those angles is less than the threshold θ_{par}, then the angle condition is satisfied. It is possible to compute the angles using goniometric functions, but those functions tend to be slow so it is better to avoid them. However, it is possible to compute only the dot product for every pair of segments. If $\mathbf{a}$ and $\mathbf{b}$ are vectors, and the angle between $\mathbf{a}$ and $\mathbf{b}$ is θ, than the following relation holds:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos\theta$$

But $|\mathbf{a}||\mathbf{b}|$ is constant because all tracts are equidistant sampled with the same sampling distance d_{sample}.

$$\mathbf{a} \cdot \mathbf{b} = C \cos\theta$$

$$\mathbf{a} \cdot \mathbf{b} > C \cos\theta_{par}$$

In our case we compute the right-hand side at the beginning and reuse this value over and over, because this will stay constant. We compute it as $d_{sample}^2 \cos\theta_{par}$. Now we can just compare the dot product of every pair of segments and compare this to this constant to find out if the angle is less than θ_{par}.

4.4 Mutual indices condition

The third and last condition to be checked by the *check* method is the mutual indices condition. For this condition, we don't only need to know p and q, but also $nn(p)$ and $nn(q)$. The simplest solution is to just search for this nearest neighbors again by iterating over all vertices of the other tract. This is a little bit expensive, because always at least $p = nn(q)$ or $q = nn(p)$ (otherwise (p, q) wouldn't be a nearest neighbor pair). So this could be optimized by just searching for the missing nearest neighbor.

It could be the case that this last nearest neighbor was already found during a previous call to the *check* method (with a different pair (p, q)). It might be possible to optimize the algorithm even further by memorizing all nearest neighbors, do a lookup whenever a nearest neighbor is needed and only search for the nearest neighbor if it isn't already computed before. However, this probably results in a overhead larger than the speed improvement.

4.5 Conclusion

While the naive approach will work, and for small datasets can be pretty fast, it won't work very well for the analysis stage of the Multi-Scale Fiber Tract Bundling technique.

This because the naive approach does have a complexity of $O(n^2m)$, where n is the total number of vertices and m the average number of vertices per tract. Here, the iteration over all possible pairs (p, q) takes $O(n^2)$ time. The distance and angle check can be done in constant time, but the mutual indices condition requires to search for a nearest neighbor on a certain tract, which takes on average $O(m)$ steps and thus the total complexity is $O(n^2m)$.

Chapter 5

Grid-Search approach

5.1 Introduction

The initial goal of my bachelor research project was to find out if the analysis stage could be made faster by implementing the construction of the similarity graph using a grid search algorithm. Grid search algorithms are a class of algorithms which are useful if you are searching for *local* data. Our data consists of vertices, which are just 3-Dimensional points. This makes it straightforward to define when two points in the dataset are local to each other or not: measure the distance between both points and compare this to some threshold value.

The analysis stage needs to find all pairs of vertices (on different tracts) matching three conditions. The first condition is the distance condition. The second and third condition deal with the orientation of the tracts. The grid search algorithm can be faster than the naive algorithm if the grid search algorithm can skip tests for vertex pairs, because due to the grid search algorithm the result of the test is known beforehand. Skipping tests means there are less computations needed and thus might lead to a performance improvement, but only if the overhead of the grid search algorithm is low enough. Here, the overhead of the algorithm is the time needed for initialization and bookkeeping. If the overhead of the grid search algorithm is larger than the computation time improvement by skipping tests, then the resulting algorithm will be slower.

5.2 Grid search algorithms

Suppose we have a set of 2-dimensional points. Given any two such points, it is possible to compute the (Euclidian) distance between those points. If we want to know all pairs of points with a distance below some threshold, we could first iterate over all points, and for every point iterate over all remaining points such that we will loop over all possible pairs of points. For every pair of points, we could compute the distance and compare this to the threshold. However, this does have a computational complexity of $O(n^2)$ (where n is the number of points).

If however the used threshold is fairly small, such that most possible point pairs do have a distance greater than this threshold, we can improve the performance by 'ordering' the points in such a way that we don't have to compute most distances. Thus, we could skip tests which might lead to a better performance but this requires that we can re-order the points and store them in a data structure which makes it easy to get only the points at or close to some location.

A possible data structure is a 2-dimensional grid (with square cells) and mapping all points to a grid cell based upon the point location. Now, given a point A, we can easily find all points B close to A because we can take the points assigned to the same or a nearby grid cell. How many grid cells we have to take depends upon the chosen grid edge size and distance threshold.

For our 2-dimensional points, we can choose some edge size and an origin of the grid, and use this to place all points into a grid cell. Some cells might contain many points and others none, but in general the average number of points in every cell depends upon the density of the set of points and the chosen edge size. A smaller edge size or a lower density will result in less points per cell, while a larger edge size or a higher density results in more points per cell. It is impossible to change the density of the given data set, but the edge size gives us a parameter we can use to alter the performance of the algorithm.

If we now have some point A and want to find all points B_i with the distance between A and B_i less than some threshold value T, then we can do this by only checking the distances between A and all points in cells which can contain points B_i with the distance between A and B_i less than T. In many cases, this means we can skip lots of distance computations. To do this fast, some bookkeeping is needed. We have to store for every vertex which grid cell it belongs to, and for every grid cell which vertices are inside the cell. This bookkeeping results in some overhead and additional memory usage.

Grid search cannot only be used with 2-dimensional data, but also higher dimensional data is possible. This makes it also possible to use the grid search method for the analysis stage of the Multi-Scale Fiber Tract Bundling technique.

Formally, the grid search algorithm is still $O(n^2)$ because it might happen that - in worst case - we are still checking all possible pairs of points. But, depending upon the data set, it might be the case that on average the complexity is less then $O(n^2)$.

In the context of DTI data, it is known that the data set will be very large and dense, and the threshold value T fairly low. Because of this, it might be the case that using a grid search algorithm for the construction of the similarity graph gives a performance improvement.

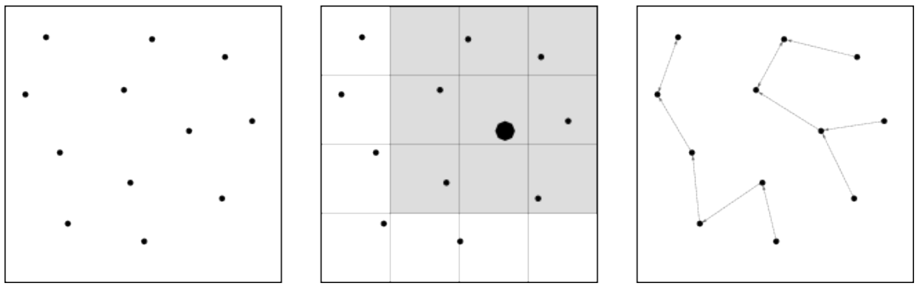


Figure 5.1: Nearest neighbor search for a set of points

5.3 The distance condition

It is clear that the distance condition for similar vertex pairs makes the grid search method attractive, because we can simply choose the value of the threshold T as the grid edge size. Since the vertex data is spatial by nature, it is also clear how to assign every vertex to a grid cell and to define the distance between any two vertices. This makes it straightforward to apply the grid search algorithm and skip distance condition tests.

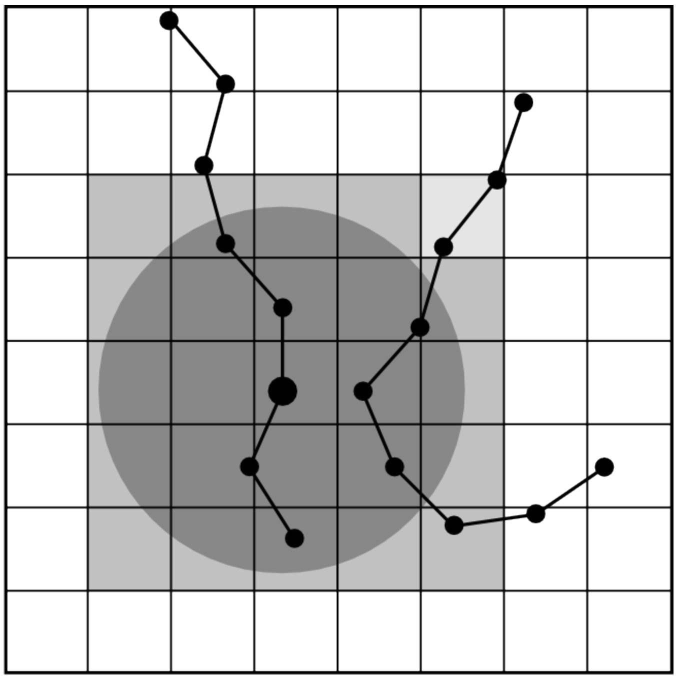


Figure 5.2: The distance condition

5.4 The angle conditions

The latter two conditions which similar vertices had to satisfy was that the angle between at least one of the corresponding incident segments had to be less then a certain threshold, and the mutual indices condition which states that, if (p, q) is a nearest neighbor pair, then the position of $nn(p)$ and q is not allowed to differ more than 1, and also the position of $nn(q)$ and p is not allowed to differ more than 1.

Those last two conditions don't leave any possibilities for optimization by using a grid search method. It might be possible to skip a few angle tests by using extensive bookkeeping, but the performance improvement won't be worth the required overhead for this bookkeeping.

5.5 Conclusion

The idea was that I should research the possibilities of using a grid search method for the last two conditions, while in the similar bachelor project research was done for the possibilities of using a grid search method for the first condition, the distance condition. Since the second and third condition don't leave much space for improvement by using a grid search method, it seems that it is better not to use a separate grid search only for the angle and mutual indices conditions because the overhead of the grid search would make matters only more worse.

However, this isn't a really satisfactory result. My research suggested that it might be possible to use another search algorithm, which is described in chapter 6.

Chapter 6

Sorted Data approach

6.1 Introduction

The grid search approach for only the second and third condition didn't give satisfactory results, but my research suggested there were some possibilities which could result in a faster algorithm. The idea is that, instead of sorting the vertices, the nearest neighbor vertex pairs are sorted. However, this requires that those nearest neighbor vertex pairs are already known, because else it isn't possible to sort the pairs.

In chapter 5 it was suggested that a grid search approach would give a good improvement for only the distance condition, but not for the last two conditions, the angle and the mutual indices condition. A first stage grid search algorithm could be used to get all nearest neighbor pairs satisfying the distance condition. Given this list of pairs, it might be possible to find all nearest neighbor pairs satisfying all three conditions. At first sight, this requires that the second and third condition are checked for all given pairs of nearest neighbor vertices satisfying the distance condition. This would require the computation of up to four angles per vertex pair, and finding the nearest neighbors of both vertices for every vertex pair. However, it seemed to be possible to do this much faster by sorting the vertex pairs before processing them.

6.2 Algorithm Overview

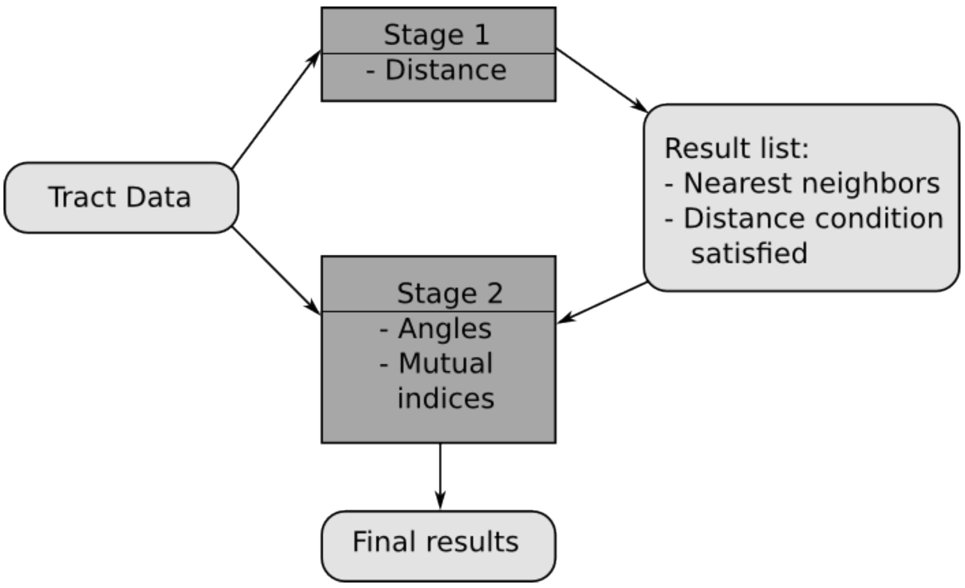


Figure 6.1: Algorithms overview

This second stage algorithm gets as input not only the original tract data, but also the results of the first stage. Those results are a single list of 5-tuples (a, i, b, j, d) with two vertices on two different tracts and their distance:

a the index of the first tract of the pair

i the index of the vertex on the first tract

b the index of the second tract of the pair (with $b > a$)

j the index of the vertex on the second tract

d the distance between the first and second vertex

Here, all a, i, b and j are integer numbers and the d are floating point numbers. The condition $b > a$ ensures that all possible pairs can be written in only one way. Without this condition, the tuple (b, j, a, i, d) would represent the same vertex pair as the tuple (a, i, b, j, d).

6.3 The distance condition

For all pairs given, it is known that the corresponding pair of vertices satisfies the distance condition, and it has to be determined if they also satisfy the other two conditions. If for some reason the distances d aren't known, then it is easily possible to recompute them for all given pairs. But because in general the first stage will already have computed this distances, it might be easier to just pass this distances instead of recomputing them.

6.4 The angle condition

Given an input list of pairs described as 5-tuples (a, i, b, j, d) it is possible to verify the angle condition in one pass by just iterating over all elements of the input list, compute the up to four different angles and if at least one of them is less than the threshold value, then the input element does satisfy the angle condition.

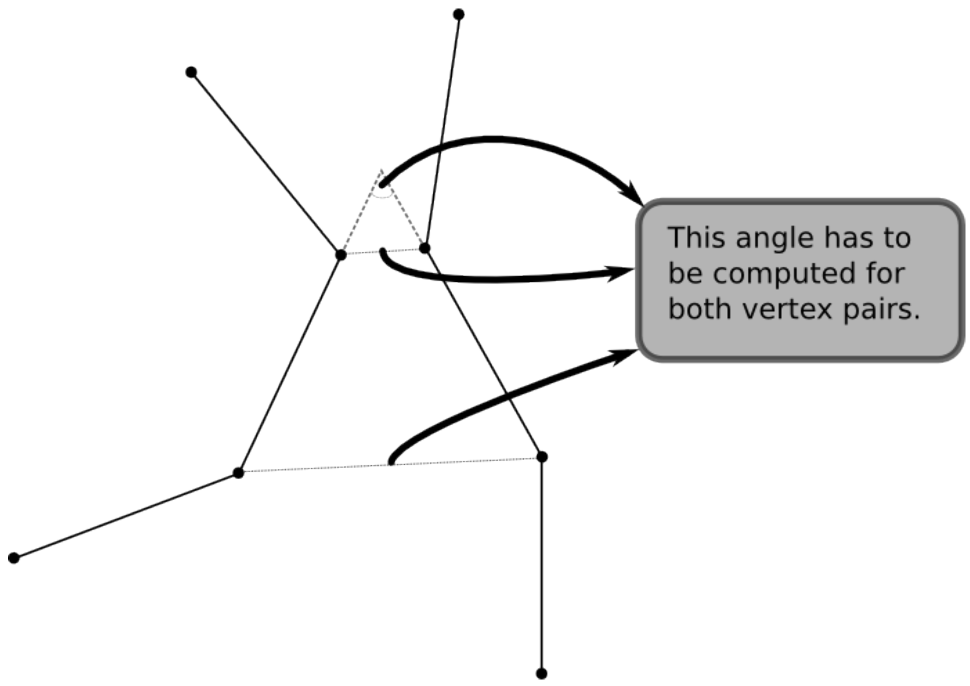


Figure 6.2: Same angle might be used for two vertex pairs

But in many cases many angles will be computed twice, for two different pairs of vertices, where the corresponding vertices are each others neighbor. It is possible to prevent this by sorting the input list before processing the elements. This is possible if and only if the input list is sorted first with respect to a, then with respect to b, if both a and b are equal then sorting is done with respect to i, if even those equal with respect to j and finally with respect to d. Thus, the fields are sorted and the order of importance of the fields is: $[a, b, i, j, d]$.

If we are processing a 5-tuple (a, i, b, j, d) then this tuple represents the vertices a_i and b_j. Now, a_i will have the incident segments $\vec{a}_{i-1}$ and $\vec{a}_i$ and b_j will have the incident segments $\vec{b}_{j-1}$ and $\vec{b}_j$. This gives us four possible incident segment pairs:

$$\begin{array}{rc} LL: & (\vec{a}_{i-1}, \vec{b}_{j-1}) \\ LR: & (\vec{a}_{i-1}, \vec{b}_j) \\ RL: & (\vec{a}_i, \vec{b}_{j-1}) \\ RR: & (\vec{a}_i, \vec{b}_j) \end{array}$$

Here, LL stands for $left\ left$, LR for $left\ right$ and so on.

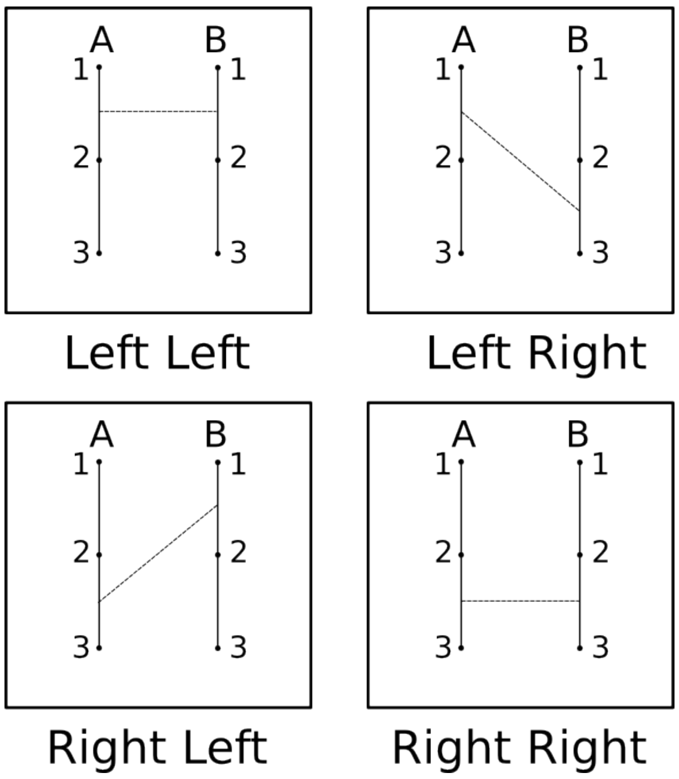
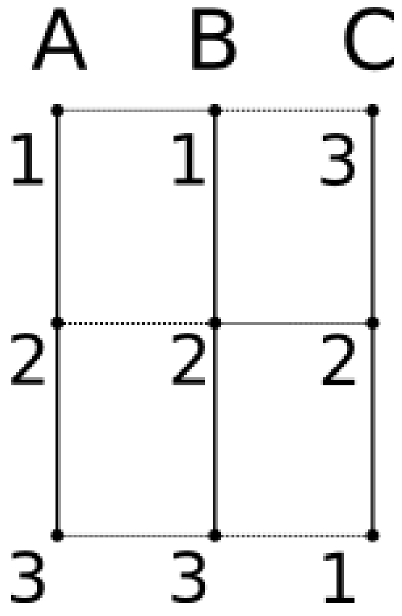
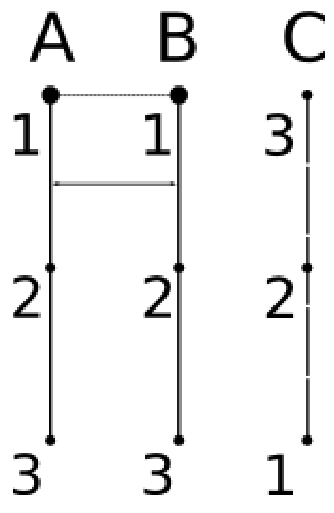


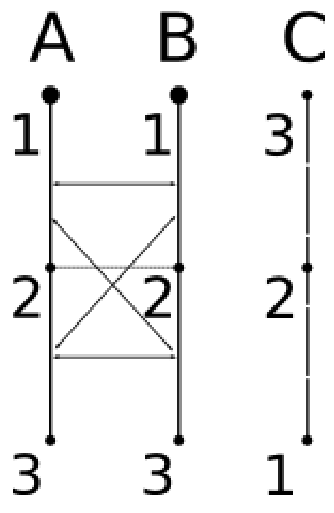
Figure 6.3: There are up to four segment pairs possible for vertex pair (a_2, b_2)



(a) Example dataset



(b) Iteration 1



(c) Iteration 2

Figure 6.4: First two iterations

In image 6.4(a) we see a tiny dataset with only 3 tracts. Each tract does have 3 vertices. The tracts are perfectly parallel, and d_{max} is chosen so that $D_{A,B} \leq D_{B,C} < d_{max} < D_{A,C}$ where $D_{X,Y}$ denotes the distance between tract X and Y (thus X and Y have to be parallel). This gives six vertex pairs satisfying the distance condition:

1.	$(A, 1, B, 1, D_{A,B})$:	a_1	b_1
2.	$(A, 2, B, 2, D_{A,B})$:	a_2	b_2
3.	$(A, 3, B, 3, D_{A,B})$:	a_3	b_3
4.	$(B, 1, C, 3, D_{A,B})$:	b_1	c_3
5.	$(B, 2, C, 2, D_{A,B})$:	b_2	c_2
6.	$(B, 3, C, 1, D_{A,B})$:	b_3	c_1

Here, the vertex pairs are sorted as described earlier.

Suppose that it isn't known that these tracts are parallel, and we want to determine if they are approximately similar. Then we will have to verify the angle condition for every vertex pair. In the first iteration we have to verify the angle condition for the first vertex pair. As can be seen in figure 6.4(b), vertex pair nr. 1 does only have one corresponding pair of segments, the so-called RR segment pair. Computing the angle of this segment pair will learn us that this angle is less than the threshold θ_{par} and thus that this angle is valid.

In the second iteration we have to verify the angle condition for the second vertex pair. Now, there are four different vertex pairs as can be seen in 6.4(c). We can compute the angle for all four vertex pairs, but then we recompute the angle we computed in the first iteration because LL_2 (the LL segment pair of the second iteration) is the same as RR_1.

In the third iteration, the same will happen: $LL_3 = RR_2$. Now we have verified all vertex pairs with one vertex on tract A and the other on tract B. In the fourth iteration, we again have only one segment pair, but this time it is the RL pair because of the 'inverse' numbering of vertices on tract C.

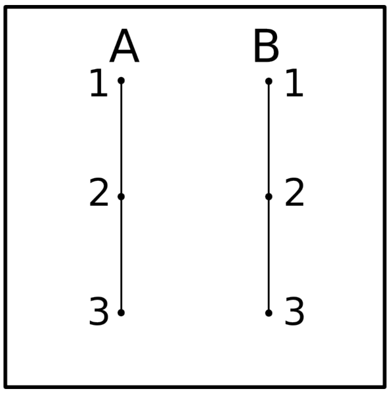
Now, in the fifth iteration it will happen that $LR_5 = RL_4$ and finally in the sixth iteration $LR_6 = RL_5$. This example learns us that we have to recompute some angle in most iterations. However in all cases this angle was computed first in the previous iteration and it was in the previous iteration either the RR or the RL pair. This happens because of the vertex pair ordering.

So, to te able to skip consecutive computations of the angle, we have to do some bookkeeping. Because the input pairs are sorted, we only have to look at the previous input element to know whether we could re-use some angle computation or not. Suppose we are processing the input element $(a_v, i_v, b_v, j_v, d_v)$, then the previous input element was $(a_{v-1}, i_{v-1}, b_{v-1}, j_{v-1}, d_{v-1})$. Now, there are two possibilities for reusing one angle computed for the previous input element:

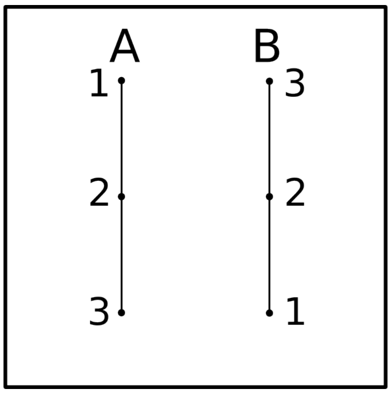
Case 1 : $a_v = a_{v-1}, b_v = b_{v-1}, i_v = i_{v-1} + 1, j_v = j_{v-1} + 1$, now LL_v is only valid if RR_{v-1} was valid.

Case 2 : $a_v = a_{v-1}, b_v = b_{v-1}, i_v = i_{v-1} + 1, j_v = j_{v-1} - 1$, now LR_v is only valid if RL_{v-1} was valid.

Those cases are almost identical. The second case happens when the indices of the tracts are ordered in opposite directions, as can be seen in the following image.



(a) Case 1



(b) Case 2

Figure 6.5: Angle reuse cases

Now, we finally have the following algorithm for verification of the angle condition:

```
vertex_pairs.sort() {Sort pairs as described above}
{Previous values of a, b, i and j, initialized to -1 for first iteration}
$(pa, pb, pi, pj) \leftarrow (-1, -1, -1, -1)$
$(ll_valid, rr_valid, rl_valid, lr_valid) \leftarrow (false, false, false, false)$
for all $(a, i, b, j, d) \in vertex\_pairs$ do
  if $a = pa \wedge b = pb \wedge i = pi + 1 \wedge j = pj + 1$ then
    $ll\_valid \leftarrow rr\_valid$ {Case 1}
  else
    $ll\_valid \leftarrow check\_segment\_pair(LL, a, i, b, j)$
  end if
  if $a = pa \wedge b = pb \wedge i = pi + 1 \wedge j = pj - 1$ then
    $lr\_valid \leftarrow rl\_valid$ {Case 2}
  else
    $lr\_valid \leftarrow check\_segment\_pair(LR, a, i, b, j)$
  end if
  $rr\_valid \leftarrow check\_segment\_pair(RR, a, i, b, j)$
  $rl\_valid \leftarrow check\_segment\_pair(RL, a, i, b, j)$
  if $ll\_valid \vee rr\_valid \vee rl\_valid \vee lr\_valid$ then
    $mark\_result(a, i, b, j)$
  end if
  $(pa, pb, pi, pj) \leftarrow (a, b, i, j)$
end for
```

6.5 The mutual indices condition

While in the naive implementation it is possible to test for the angle condition in one run without sorting first, this isn't possible for the mutual indices condition. However, this can be done in a similar way as with the distance condition. Again, the input pairs have to be sorted, but this time the order of importance of the fields is: $[a, b, d, i, j]$. By sorting it this way, we ensure that the vertex pairs are ordered so that all vertex pairs between the same tract pair are grouped together. Moreover, for each tract pair we now have all corresponding vertex pairs, sorted by distance.

Suppose we now want to check the mutual indices condition for all given vertex pairs in one loop. We know that the input vertex pairs are all nearest neighbor vertex pairs satisfying the distance condition. Because of the chosen ordering, the algorithm will handle all vertex pairs corresponding with a pair of tracts after each other. When processing tract A and B, we can easily hold two mappings, one mapping the vertices on A to their nearest neighbor on B, and the other mapping the vertices on B to their nearest neighbor on A.

However, it still seems to be necessary to find the nearest neighbors to fill in these mappings. But this isn't needed, because the vertex pairs are sorted with respect to the distance too. It is important that this sorting is done in ascending order. We already know that all input vertex pairs are nearest neighbor pairs, so we only have to find out if the vertex pair is a nearest neighbor pair because the vertex on B is a nearest neighbor of the vertex on A, or the other way around the vertex on A is a nearest neighbor of the vertex on B. Or maybe even both are a nearest neighbor of the other.

Because of the sorting with respect to the distance d, we will process the vertex pairs closest together first. So we can just update our nearest neighbor mappings every iteration, and clear the mappings when we start processing a new pair of tracts. Again assume that we are processing the input element $(a_v, i_v, b_v, j_v, d_v)$ and the previous input element was $(a_{v-1}, i_{v-1}, b_{v-1}, j_{v-1}, d_{v-1})$.

For every pair of tracts we keep two arrays for the nearest neighbor mappings:

- *nna* with the same length as the current A tract, initialized to -1 (meaning *unknown*).
- *nnb* with the same length as the current B tract, also initialized to -1.

When at some iteration $a_v \neq a_{v-1}$ or $b_v \neq b_{v-1}$ then both *nna* and *nnb* are recreated (due to possibly changed lengths) and again initialized to -1 (unknown) values.

Now, it is possible to check if the current element satisfies the mutual indices condition. If $nna[i] \neq -1$ and $|nna[i] - j| > 1$, then the mutual index condition check failed. The same happens when $nnb[j] \neq -1$ and $|nnb[j] - i| > 1$. When the mutual index condition is satisfied, then *nna* and *nnb* have to be updated for the next iterations. If $nna[i] = -1$, then there isn't yet a value known as the nearest neighbor of a_i on the B tract, so this can be set to b_j but only the value j has to be stored in $nna[i]$. The same should be done for $nnb[j]$ and i.

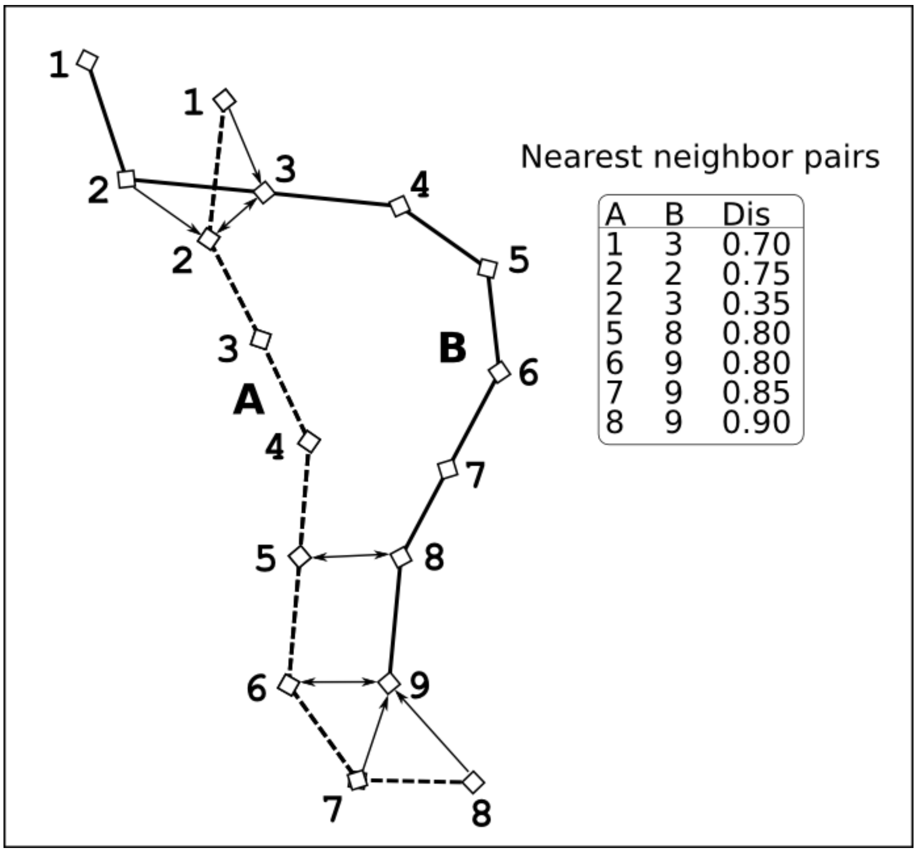
This gives us finally the following algorithm:

```
vertex_pairs.sort() {Sort pairs as described above}
{Previous values of a, b, i and j, initialized to -1 for first iteration}
$(pa, pb, pi, pj) \leftarrow (-1, -1, -1, -1)$
for all $(a, i, b, j, d) \in vertex\_pairs$ do
  if $a \neq pa \vee b \neq pb$ then
    {$array(x)$ creates a new integer array of length x}
    $nna \leftarrow array(tract\_length(a))$
    {set_array_values(A, V) sets all values of the given array A to V}
    $set\_array\_values(nna, -1)$
    $nnb \leftarrow array(tract\_length(b))$
    $set\_array\_values(nnb, -1)$
  end if
  if $(nna[i] = -1 \vee |nna[i] - j| \leq 1) \wedge (nnb[j] = -1 \vee |nnb[j] - i| \leq 1)$
  then
    $mark\_result(a, i, b, j)$
    if $nna[i] = -1$ then
      $nna[i] \leftarrow j$
    end if
    if $nnb[j] = -1$ then
      $nnb[j] \leftarrow i$
    end if
  end if
  $(pa, pb, pi, pj) \leftarrow (a, b, i, j)$
end for
```

See for example the dataset in figure 6.6. There are two tracts, A (dashed) and B (solid). The first stage algorithm has found 7 nearest neighbor pairs. If we sort those pairs as described above, then the third pair will become the first while the other pairs don't move.

In the first iteration, $a \neq pa$ and $b \neq pb$, so nna and nnb are created, nna has length 8, and nnb has length 9. In this iteration $i = 2$ and $j = 3$. Both $nna[i] = -1$ and $nnb[j] = -1$ so this first pair is marked as a result, the value of j is assigned to $nna[i]$ and the value of i is assigned to $nnb[j]$.

In the second iteration, $i = 1$ and $j = 3$. Now $nnb[j] = 2$, however $|1 - 2| \leq 1$ so this is still marked as a result. In the third iteration the value of $nna[i]$ won't equal -1 but the absolute value of the difference is again less then 1.



A	B	Dis
1	3	0.70
2	2	0.75
2	3	0.35
5	8	0.80
6	9	0.80
7	9	0.85
8	9	0.90

Figure 6.6: Example dataset

In the following three iterations the pair will also be accepted as a result. However in the seventh and last iteration we will have this situation: $i = 8$, $j = 9$, $nna[i] = -1$ and $nnb[j] = 6$. Here, the absolute difference between i and $nnb[j]$ is too large and thus the last pair will be rejected.

6.6 Conclusion

The sorted data algorithm does two times sort the input data. With a good sorting algorithm this can be done in $O(n \ log \ n)$ time. It also does two iterations over the input list, which both take $O(n)$. Here n is the number of elements in the input list. Thus the total algorithm will only take $O(n \ log \ n)$ time, which is at least better than the $O(n^2)$ of the naive algorithm, and also better than the grid search approach. While this algorithm does only a little bit bookkeeping, it can skip many of the computations the naive algorithm otherwise would do.

However, the price of using this algorithm is that the vertex pairs have to be sorted twice which is an additional overhead. Inspection of the time the algorithm spent on this sorting learned that approximately 25% of the running time (without file I/O) was used for sorting, where the first sorting (for the angle condition) required twice as much time as the second (for the mutual indices condition) because this second time the vertex pairs failing the angle condition where removed from the input.

Chapter 7

Test results

The speeds of the naive and the sorted data algorithms were measured and compared. The algorithms ran on a HPC cluster node with 24 Opteron 2.6 Ghz cores and 128 GB memory. However, none of the algorithms was designed to work with multiple cores, so only one 2.6 Ghz core was used.

# of pairs	Naive	Sorted Data
50710224	19m 43s	3m 14s
14518582	5m 15s	1m 1s
4190030	1m 42s	18s

For testing, a variation on the naive algorithm was used which did, like the Sorted Data approach, run over all input pairs and checked them using the *check* function. However, this time the *check* function didn't verify the distance condition, since this was already satisfied as a precondition. This naive algorithm did an $O(n)$ search for the nearest neighbors to find the absolute differences used by the mutual indices condition. The optimized version doesn't have to do this search, In combination with leaving out repeated angle computations, this results in the speedup of four to six times.

Chapter 8

Conclusions

In this bachelor project I considered the possibilities of using a grid search method in the analysis stage of the Multi-Scale Fiber Tract Bundling algorithm as described in [3], but only for the last two of the three conditions for similar vertices. It turned out that the grid search wouldn't help in the case that only the last two conditions had to be verified, but that a grid search approach is very useful for the first stage (which checks only the distance condition). However, a faster solution was possible by sorting the results of the first stage prior to processing them in the second stage. This did give the best results.

It is possible to join the first and second stage of the analysis into one algorithm. However, this requires that the first stage outputs the results in an order suitable for the second stage. This can be either the ordering used for the mutual indices condition or alternatively the one for the angle condition. It could then test the mutual indices condition (or in the alternative case the angle condition) immediately after the nearest neighbor pair has been found by the first stage algorithm. However, this ordering then isn't suitable for the optimized test of the remaining condition. One option is to do this remaining condition at the end, when the first stage is done and sorting is possible, or to do this immediately after the first condition, but in an suboptimal way (because of the wrong ordering). Further research in this area is possible and might give an algorithm with the advantages of both the first stage (as researched by Jorn v.d. Beek) and the second stage as described in this thesis.

Bibliography

[1] Aaron Filler, M.D., Ph.D. *Magnetic Resonance Neurography and Diffusion Tensor Imaging: Origins, History, and Clinical Impact of the First 50000 Cases with an Assessment of Efficacy and Utility in a Prospective 5000-Patient Study Group*, chapter 6. Congress of Neurological Surgeons, 2008.

[2] F Dargi, M A Oghabian, A Ahmadian, H Zadeh, M Zarei, and A Boroomand. Modified fast marching tractography algorithm and its ability to detect fibre crossing. *Conf Proc IEEE Eng Med Biol Soc*, 2007:319–22, 2007.

[3] Maarten H. Everts, Henk Bekker, Tobias Isenberg, and Jos B.T.M. Roerdink. Exploration of the Brain's White Matter Structure through Visual Abstraction and Multi-Scale Fiber Tract Bundling.

[4] M. Kubicki, H.-J. Park, C.-F. Westin, P. Nestor, R. Mulkern, S. E. Maier, M. Niznikiewicz, E. Connor, J. Levitt, M. Frumin, R. Kikinis, F. A. Jolesz, R. McCarley, and M. E. Shenton. DTI and MTR abnormalities in schizophrenia: Analysis of white matter integrity. *NeuroImage*, 26:1109–1118, 2005.

[5] Marek Kubicki, Robert McCarley, Carl-Fredrik Westin, Hae-Jeong Park, Stephan Maier, Ron Kikinis, Ference A. Jolesz, and Martha E. Shenton. A review of diffusion tensor imaging studies in schizophrenia. *Journal of Psychiatric Research*, 41:15–30, 2007.

[6] J.M.J. Murre and D. P. F. Sturdy. The connectivity of the brain: Multilevel quantitative analysis. *Biological Cybernetics*, 73:529–545, 1995.

[7] C H Salmond, D K Menon, D A Chatfield, G B Williams, A Pena, B J Sahakian, and J D Pickard. Diffusion tensor imaging in chronic head injury survivors: correlations with learning and memory indices. *NeuroImage*, 29(1):117–24, 2006.