Automatic Junction Detection of Tubular Structures

Release 0.1

Guanglei Xiong[1], Lei Xing[2] and Charles Taylor[3]

Mar 04, 2009

[1]Biomedical Informatics Program, Stanford University
[2]Department of Radiation Oncology, Stanford University
[3]Department of Bioengineering, Stanford University

Abstract

Junctions of tubular structures (vasculature, trachea, neuron, etc) in medical images are critical for the topology of these structures. Identification of them is helpful in many applications. For example, quantification of geometric vascular features, registration of trachea movement due to respiration, tracing of neuron path. However, manual extraction of junctions can be tedious, time-consuming, and subject to operator bias. In this paper, we propose a novel method to detect them automatically and describe how to implement it in ITK framework. The input is a 2D/3D binary image that can be obtained from any segmentation techniques. The output will be positions of junctions and their sizes. There are only two parameters which need to be set by the user. We provide here the implementation as a ITK class: `itk::JunctionDetectionFilter`.

Contents

1 Introduction **1**

2 Design **2**

3 Implementation **3**

4 Usage and Examples **4**

5 Software Requirements **6**

1 Introduction

Junctions of tubular structures in medical images are critical for their topology. These structures include vasculature, trachea, neuron, etc. Identification of them are helpful in many applications. For example, the junctions of vessels can be used to define vascular trees and quantify the lengths of vessel segments.

Tracking of junctions in trachea enables more robust registration of movement due to respiration. Junctions make the tracing of the paths of neuron easier that can be useful to study their growth. However, manual extraction of junctions can be tedious, time-consuming, subject to operator bias, and intractable when many are present. To our knowledge, there is no free and open-source software that are readily available for this task. In this paper, we propose a novel method to detect them automatically and describe how to implement it in ITK framework [1]. In the end, we introduce how to use our implementation and demonstrate our results.

2 Design

Junctions in tubular structures are not easy to detect for several reasons. First, they are topologically distinct from tubular structures. So detection of the latter does not generally yield the former unless specific treatment is considered. Second, the junctions are of various shapes which can be bifurcations, trifurcations or even more branches. For one bifurcation, it may have a Y-shape or a T-shape. Furthermore, it is usually necessary to detect junctions of different sizes simultaneously. We designed our method to take these challenges into account.

The basic idea of our approach is simple. For every voxel in the image as shown in Fig. 1, we place a hollow sphere centered at the voxel and with inner radius r_{in} and outer radius r_{out}. Then, we count the number of connected components inside the hollow sphere. Clearly, if there are three or more components, the voxel should be a junction point. For example in Fig. 1, hollow spheres 1 and 6 have two and one components, respectively and their center are not junctions. But hollow sphere 3 has three components and its center is a bifurcation point. As you can imagine, trifurcations in general lead to four connected components.

We further develop the basic idea by introducing mechanisms that better handle the difficulties mentioned before. First, in order to handle junctions of different sizes, inner and outer radii are dependent on the distance of the voxel to the wall of the object d_{w}. That is, $r_{\mathrm{in}} = \alpha d_{\mathrm{w}}$ and $r_{\mathrm{out}} = \beta d_{\mathrm{w}}$, where $\alpha < \beta$ are user-defined parameters. This definition of radii result in two effects. One is the size of a hollow sphere will adapt to the size of tubular structures or junctions, e.g. hollow spheres 3 and 5 in Fig. 1. The other is voxels near the centerline are tested by a larger sphere than voxels near the wall, e.g. hollow spheres 1 and 2. This will lead to fewer voxels for each junction and push them into the center of tubular structures. The second mechanism is we employed the geodesic distance in our distance calculation rather than simple Euclidean distance as shown in Fig. 1. This includes the distance to the wall and the distance that is used to see which voxels are inside a hollow sphere. One advantage of doing this is to avoid the situation that branches intersect the hollow sphere of the voxel of interest by Euclidean distance but they are actually distant by geodesic distance, otherwise will result in a false junction point. The third mechanism is that we have minimal number of voxels $n_{\min}$ in the hollow sphere in order for the voxel of interest to be qualified for a junction point. It is a practical consideration that comes from small number of voxels often result in incorrect number of connected components because of the discrete nature of images. Finally, after all the junction points are detected. We need to classify them and select a representative point for all points that belong to each junction. Distance-based clustering is used to separate them into different junctions. Basically, two points which are near each other are assigned into one junction. The representative point for each junction is the point with the largest distance to the wall among all points assigned to that junction. The distance to the wall for the representative point indicates the size of the bifurcation.

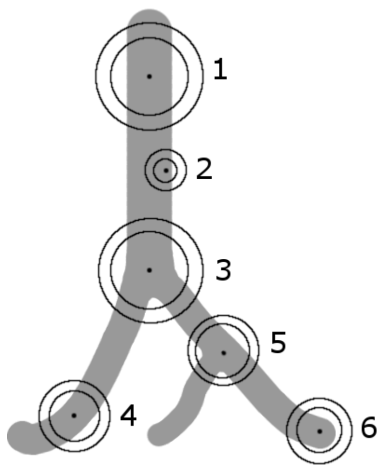
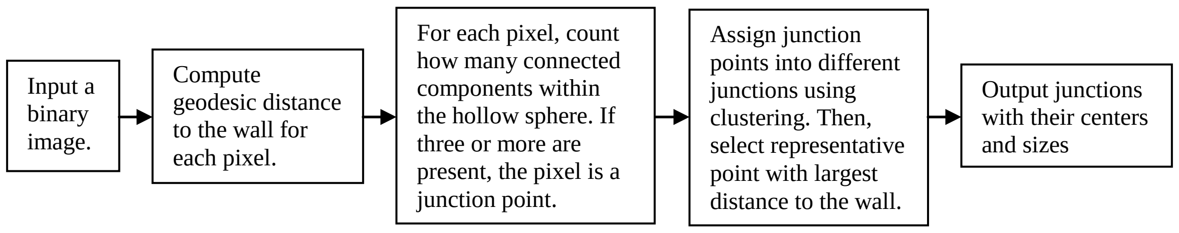


Figure 1 The basic idea of our bifurcation detection method is to test how many connected components in each hollow sphere.

The flowchart of our method is summarized in Fig. 2.



Input a binary image.

Compute geodesic distance to the wall for each pixel.

For each pixel, count how many connected components within the hollow sphere. If three or more are present, the pixel is a junction point.

Assign junction points into different junctions using clustering. Then, select representative point with largest distance to the wall.

Output junctions with their centers and sizes

Figure 2 The flowchart of our junction detection method.

3 Implementation

We implemented the method as a ITK class: `itk::JunctionDetectionFilter`, which is subclass of `itk::ImageToImageFilter`. It is templated by the input image type and controlled by three functions:

- **[Set/Get]InnerRadius**: define α .
- **[Set/Get]InnerRadius**: define β .
- **[Set/Get]MinNumberOfPixel**: define $n_{\min}$.

The input image should be a binary image with zero background and nonzero foreground. The output image is an image with junctions drawn by their sizes and marked by their indices. The information of junctions can also be queried by a std::map: JCLabelMapType, defined as:

```
typedef std::pair< IndexType, float >             JCLabelPairType;
typedef std::map< long, JCLabelPairType >         JCLabelMapType;
```

Each element in the map is a junction with the junction index as its key and the pair of center pixel index and size as its value.

The core of our method is in the function of `GenerateData()`. We made use of `itk:: SignedMaurerDistanceMapImageFilter` to compute distance transform of the input image and `itk:: SampleMeanShiftClusteringFilter` to cluster junction points into junctions.

4 Usage and Examples

It is very easy to use the class. It behaves like a common filter which has one input and one output.

First, a 3D image of the Circle of Willis with detected junctions is shown in Fig. 3.

Total number of detected junctions: 26 (true positive: 26, false positive: 0, false negative: 1).

The code snippet is

```
typedef itk::JunctionDetectionFilter<ImageType>   DetectorType;
DetectorType::Pointer detector = DetectorType::New();
detector->SetInnerRadius(2.0);
detector->SetOuterRadius(3.0);
detector->SetMinNumberOfPixel(16);
detector->SetInput(inputImage);
detector->Update();
```

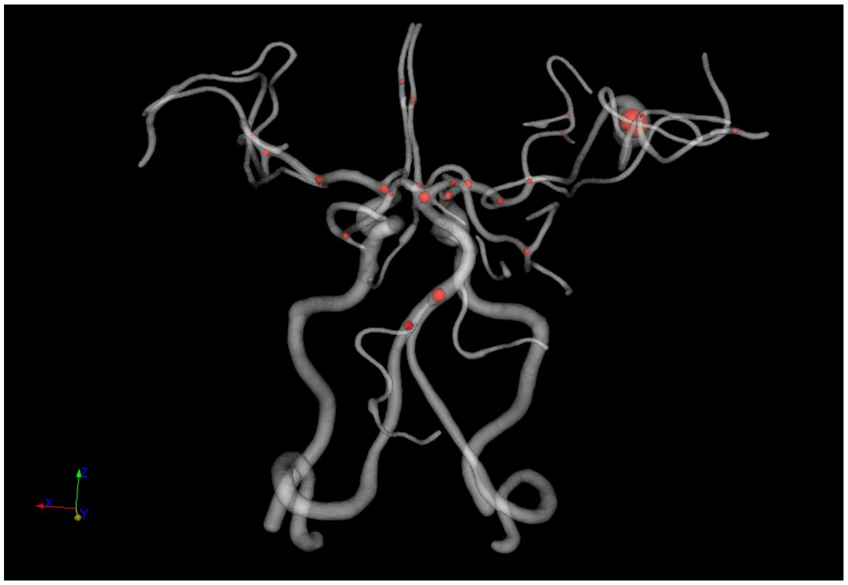


Figure 3 The Circle of Willis and its junctions.

Second, a 3D image of trachea with detected junctions is shown in Fig. 4.

Total number of detected junctions: 15 (true positive: 14, false positive: 1, false negative: 0).

The junctions are detected using the same code and parameters.

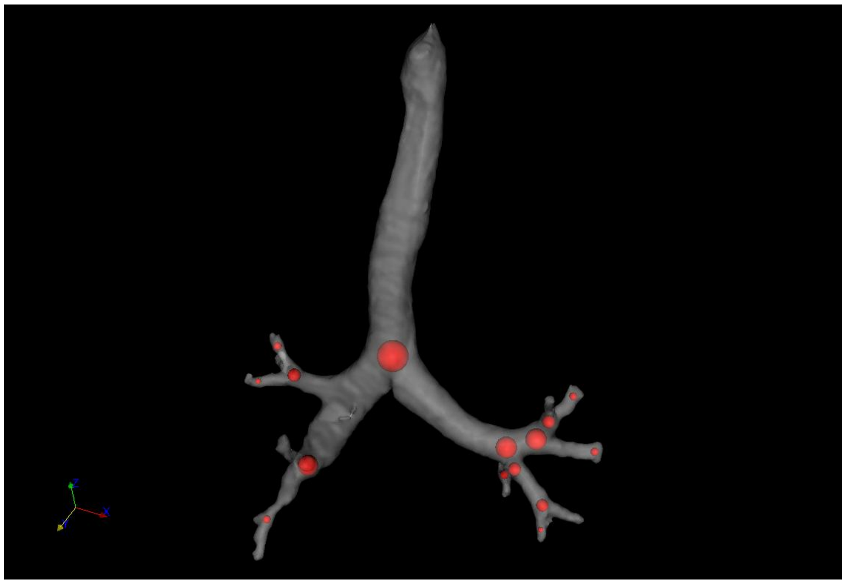


Figure 4 The trachea with its junctions.

Finally, we tested the method on a 2D neuron image as seen in Fig. 5.

Total number of detected junctions: 65 (true positive: 62, false positive: 3, false negative: 2).

The same code and the parameters were used except $n_{\min} = 6$.

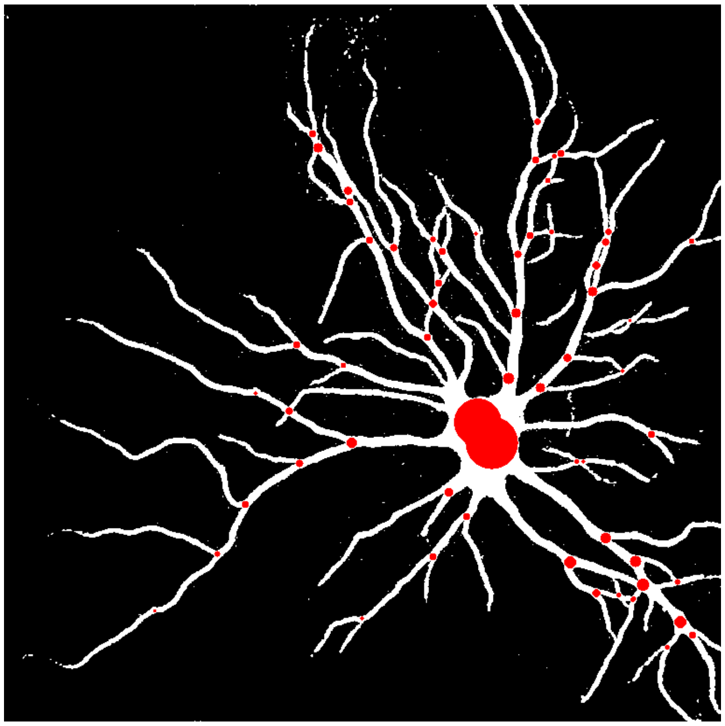


Figure 5 The neuron with its junctions.

5 Software Requirements

This paper has described a method of automatic junction detection. The method requires a minimal number of parameters and works both in 2D and 3D. The implementation is ready to be used within the ITK framework. In the future work, we hope to speed up the method by parallelism. For suggestions or bugs, feel free to contact us[1].

6 Software Requirements

You need to have the following software installed:

- Insight Toolkit 3.8.0
- CMake 2.4
- Visualization Toolkit 5.2.1 (to run the 3D test)

Acknowledgement

We would like to thank Nan Xiao for providing the segmentation of the Circle of Willis. Guanglei Xiong was supported by a Stanford Graduate Fellowship.

Reference

[1] L. Ibanez and W. Schroeder. *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-10-6, http://www.itk.org/ItkSoftwareGuide.pdf, 2003.

[1] Corresponding author: Guanglei Xiong: guangleixiong at gmail.com