Implementation of weighted Dijkstra's shortest-path algorithm for n-D images

Release 1.00

Lior Weizman[1], Moti Freiman[1] and Leo Joskowicz[1]

January 29, 2009

[1]School of Engineering and Computer Science, The Hebrew University of Jerusalem, Israel

Abstract

This paper describes the ITK implementation of a shortest path extraction algorithm based on graph representation of the image and the Dijkstra shortest path algorithm. The method requires the user to provide two inputs: 1. path information in the form of start, end, and neighboring mode, the form of which path is allowed to propagate between neighboring pixels, and 2. a weighting function which sets the distance metric between neighboring pixels. A number of perspectives for choosing weighting functions are given, as well as examples using real images. This paper can also serve as an example for utilizing the Boost C++ graph library into the ITK framework.

Keywords: minimal path, centerline, vessel segmentation, ITK, boost

Contents

1 Introduction

Shortest path is a useful algorithm for many applications including medical image analysis. In ITK only fast-marching based shortest path is published [5]. In this work we present a shortest path algorithm based

on the graph representation of an image. This representation uses the "Boost" open source library under the ITK framework. In this framework, variety of graph-based algorithms can be easily implemented on n-D images. The main principle of utilizing the "Boost" library under the ITK framework can be understood by reviewing the implementation of the main filter presented in this paper.

We use the "Boost" open source library in order to represent the image as a graph, and to find the shortest path between two voxels in the image. Each voxel in the image is represented as a node in the graph. Adjacent nodes are connected by edges. The distance between two adjacent nodes is represented by the weight of their connecting edge. The definition of the optimal weighting function is case dependent and should be set by the user, along with the start and end voxels of the path. W describe two possible weighting functions further in this paper. The shortest path between the start and the end voxels is found using Dijkstra's shortest-path algorithm [4].

We use the following notation. Let $G = (V,E)$ be the image graph, where $V = \{v_1,...,v_n\}$ are the graph nodes (one v_i per voxel) with their associated voxel intensity values, $I(v_i)$. The nodes v_s and v_f are the user-defined start and end seed voxels. $E = \{(v_i,v_j)\}$ are the graph edges, for all the pairs of neighboring voxels. Each node has 4 or 8 neighbors for 2D images, or 6 or 26 neighbors for 3D images. Each edge has a weight associated to it, $w(v_i,v_j)$.

The shortest path is the sequence of edges connecting v_s to v_f for which the sum of its edge weights is minimum. We use Dijkstras shortest-path algorithm whose worst-case complexity is $O(n^2)$, where n is the number of image voxels.

The definition of the weighting function of the edges depends with the characteristics of the shortest path that has to be found. In this paper, we present an example of finding the shortest path within a vessel in a CT image, along with two possible weighting functions. The first example is a simple gradient based weighting function. The second example is a hybrid weighting function, which takes into account the following requirements: a) The resulting path should not cross borders in the image, b) The resulting path should be straight as possible, and c) The resulting path should be with homogeneous intensity. These requirements are expressed by the following edge weighting function terms:

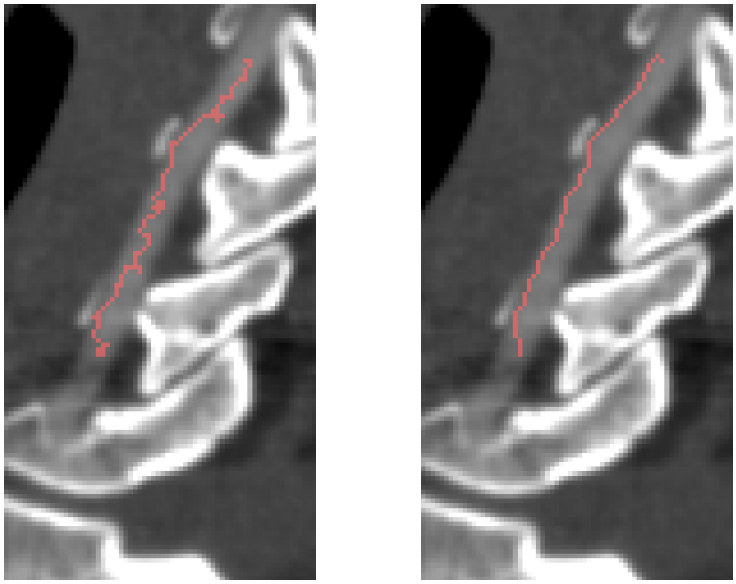
1. **Intensity difference** - the squared difference of the edge voxel intensity values: $(I(v_i)-I(v_j))^2$. Since its value is large at boundary crossings, it prevents the path from crossing boundary areas.
2. **Path smoothness** - the angle between the edge voxels gradient directions: $acos(\frac{\nabla v_i \cdot \nabla v_j}{|\nabla v_i| \cdot |\nabla v_i|})$. This term prevents edges with large gradient differences to be added to the path.
3. **Seed deviation intensity difference** - the sum of the relative squared differences of the seeds and edge voxel intensity values: $(I(v_j)-I(v_f))^2+(I(v_j)-I(v_s))^2$. This term prevents the edges in the path from diverging too much from the intensity values of the user-selected seed points.

The weighting function can be replaced or modified in the code below, to meet the requirements of any other specific application.

2 Proposed class and implementation

2.1 Overview

This project implements a number of auxiliary functions. The main filter to be employed by the user is `itk::ShortestPathImageFilter`. This filter provides the functionality of finding the shortest path in an



(a) (b)

Figure 1: The shortest path in a 2D vessel image while using two different weighting functions: (a) `itk::WeightSimpleMetricCalculator` and (b) `itk::WeightGradAngleMetricCalculator`.

image between two given start and end voxels. To define the weighting function between two neighboring voxels, we define the abstract class `itk::WeightMetricCalculator`. An instance of a sub-class derived from this class is required in order to feed `itk::ShortestPathImageFilter` with the required weighting function.

The `itk::ShortestPathImageFilter` represents the image as a graph, and finds the shortest path between the start and end points using the Dijkstra shortest path algorithm. The graph representation and the path extraction is done using the "Boost" C++ library. Therefore, a proper installation of the "Boost" library [3] is required on the user's machine.

2.2 Weighting function

Choosing an appropriate weighting function is the most important input required by the user. This is done by extending the abstract class `itk::WeightMetricCalculator` into an appropriate sub-class, while implementing the abstracts methods `GetEdgeWeight` and `Initialize` (and optionally adding more methods, as necessary).

For example, the class `itk::WeightSimpleMetricCalculator` is derived from `itk::WeightMetricCalculator` and defines a simple gradient based weighting function. Another example of defining a weighting function is the class `itk::WeightGradAngleMetricCalculator`. This class implements a hybrid weighting function that is a sum of a simple gradient based weight, the angle between the gradients of the base voxel and its neighbor and the sum of the relative squared differences of the seeds and edge voxel intensity values. Figure 1 presents the obtained path with two different weighting functions.

The main class, `itk::ShortestPathImageFilter` requires an instance of a class derived from `itk::WeightMetricCalculator` used to compute the weighting function. In the ex-

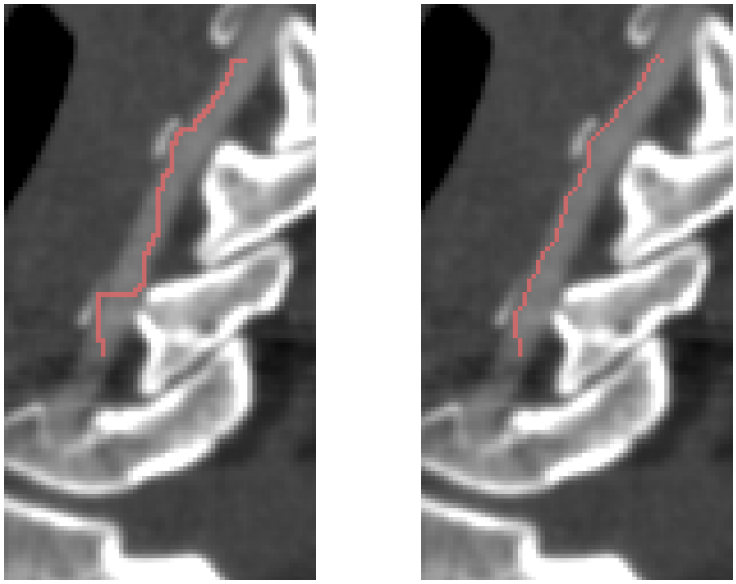
ample below, an instance of itk::ShortestPathImageFilter is fed by an instance of itk::WeightGradAngleMetricCalculator.

```
//typedefs
typedef signed short  PixelType;
typedef unsigned char OutputPixelType;
const unsigned int    Dimension = 2;
typedef itk::Image<PixelType, Dimension > ImageType;
typedef itk::Image<OutputPixelType, Dimension > OutputImageType;
typedef itk::ShortestPathImageFilter <ImageType, OutputImageType>
             itkShortestPathImageFilterType;

// Class itk::WeightGradAngleMetricCalculator is derived from
//itk::WeightMetricCalculator and defines a hybrid weighting
//function
typedef itk::WeightGradAngleMetricCalculator<ImageType> MetricType;
typedef itk::ImageFileReader< ImageType > ReaderType;

// Instantiating and initializing metric object.
MetricType::Pointer metric = MetricType::New();
metric->SetImage(reader->GetOutput());
metric->SetStartIndex(start);
metric->SetEndIndex(end);
metric->Initialize();
metric->SetSigma(5.0);

// Instantiating itk::ShortestPathImageFilter object and assigning
//the metric object
itkShortestPathImageFilterType::Pointer
    dijkstra=itkShortestPathImageFilterType::New();

// Set the metric instance for the path
dijkstra->SetMetric(metric);
```

2.3 Path information

Dijkstra shortest path extraction is a semi-automatic segmentation method - the user is required to provide start and end points. Our implementation also enables the user to select the possible path propagation in the image. Setting the neighboring mode to full neighbors modes (by calling the method SetModeToFullNeighbors()), will allow the path to propagate from an initial voxel to each and every one of its neighbors (i.e. to one of its 8 or 26 neighbors in the 2D or 3D cases, respectively). Setting the neighboring mode to non-full neightbors mode (by calling the method SetModeToNonFullNeighbors()) will allow the path to propagate from a voxel only to one of its non-diagonal neighbors (i.e. to one of its 4 or 6 neighbors in the 2D or 3D cases, respectively). In the example below, the itk::ShortestPathImageFilter is is fed with the input image, start and end voxels and the neighboring mode. The last two lines in the code below activates the filter and outputs the result into the output image pointer.

```
//Define the input image
```



(a) (b)

Figure 2: The shortest path in a 2D vessel image while using two different neighboring modes: (a) 4-neighbors mode and (b) 8-neighbors mode.

```
2  dijkstra->SetInput (reader->GetOutput());
3
4  //Define the start and end points
5  ImageType::IndexType start, end;
6  start[0] = 21;
7  start[1] = 76;
8  end[0]   = 52;
9  end[1]   = 12;
10 dijkstra->SetStartIndex(start);
11 dijkstra->SetEndIndex (end);
12
13 //Define the neighboring mode
14 dijkstra->SetModeToFullNeighbors();
15 dijkstra->Update();
16 OutputImageType::Pointer resVolume = dijkstra->GetOutput();
```

Figure 2 presents the effect of the neighboring mode selection on the extracted path.

3 Examples

In this section, wepresent two weighting functions, along with a proper usage of the main filter, `itk::ShortestPathImageFilter` to extract the shortest path from an image.

To define a new weighting function, we define a new class derived from the abstract class `itk::WeightMetricCalculator`. It implements the abstracts methods `GetEdgeWeight` and `Initialize` (and can optionally implement more methods, as necessary).

The method `GetEdgeWeight` inputs two iterators as parameters. The first one points to the base voxel and is of type `itk::ShapedNeighborhoodIterator< TInputImageType >`. The second points to one of the neighbors of the base voxels, therefore it is of type `itk::ShapedNeighborhoodIterator< TInputImageType >::Iterator`. Examples for proper implementation of this method are given below.

3.1 Gradient based weighting function

The class `itk::WeightSimpleMetricCalculator` is derived from `itk::WeightMetricCalculator` and implements a simple gradient based weighting function, by implementing the function `GetEdgeWeight`, as follows:

```
template<class TInputImageType>
double
WeightSimpleMetricCalculator<TInputImageType>
::GetEdgeWeight (const itkShapedNeighborhoodIteratorType &it1,
    const itkShapedNeighborhoodIteratorforIteratorType &it2)
{
    double a,b;
    a=it1.GetCenterPixel();
    b=it2.Get();
    return (b-a)*(b-a);

}
```

3.2 Hybrid weighting function

The class `itk::WeightGradAngleMetricCalculator` is derived from `itk::WeightMetricCalculator` and implements a hybrid weighting function. The components of this weighting functions were previously presented in Section 1 .

To compute the second term of weighting function, a gradient image has to be created. Therefore, two data-members that hold the gradient image and the standard deviation of the gradient filter are defined, and the method `Initialize` is updated to build the gradient image using the class `itk::GradientRecursiveGaussianImageFilter <TInputImageType>` as follows:

```
template<class TInputImageType>
void WeightGradAngleMetricCalculator<TInputImageType>
::Initialize()
{
 GradientRecursivePointer grad =
    itkGradientRecursiveGaussianImageFilterType::New();
 grad_radius.Fill(1);
 grad->SetInput (m_Image);
 grad->SetSigma (m_Sigma);
 grad->Update();
 SetGradImage(grad->GetOutput());
}
```

To compute the third term of the weighting function, two additoinal data-members that hold the indices of the start and end voxels of the path (as was entered by the user) are added, and the weighting function is defined as follows:

```
template<class TInputImageType>
double
WeightGradAngleMetricCalculator<TInputImageType>
::GetEdgeWeight (const itkShapedNeighborhoodIteratorType &it1,const
    itkShapedNeighborhoodIteratorforIteratorType &it2)
{
 IndexType &index1=it1.GetIndex();
 unsigned int i = it2.GetNeighborhoodIndex();
 PixelType a,b;
 a=it1.GetCenterPixel();
 b=it2.Get();
 double iDifference=(b-a)*(b-a);
 grad_radius.Fill(1);
 itkShapedNeighborhoodGradIteratorType grad_it1(grad_radius,
    m_GradImage, m_GradImage->GetRequestedRegion() );
 grad_it1.SetLocation(index1);
 GradPixelType a_grad=grad_it1.GetCenterPixel();
 GradPixelType b_grad=grad_it1.GetPixel (i);
 double pSmoothness=fabs((acos((a_grad*b_grad)/(a_grad.GetNorm()*
    b_grad.GetNorm())) / M_PI))*10000;
 PixelType startValue=m_Image->GetPixel(m_StartIndex);
 PixelType endValue=m_Image->GetPixel(m_EndIndex);
 double sIDifference=(b-startValue)*(b-startValue)+
                     (b-endValue)*(b-endValue);
 double weight=iDifference+pSmoothness+sIDifference;
 return weight;
}
```

The user can replace the weighting function with one that fits the specific requirements of the application.

We include the file `itkDijkstraFilterTest2D.cpp` which produces the 2-D results presented in this paper. Examples of finding the shortest path in a 3-D image can be found in the included file `itkDijkstraFilterTest3D.cpp`.

4 Software Requirements

You need to have the following software installed:

- Insight Toolkit 2.4 [2].
- CMake 2.2 [1].
- Boost C++ libraries 3.7 [3].

References

[1] http://www.cmake.org. 4

[2] http://www.itk.org. 4

[3] http://www.boost.org. 2.1, 4

[4] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 1

[5] D. Mueller. Fast marching minimal path extraction in ITK. *Insight Journal*, 2008. http://www.insight-journal.org/browse/publication/213. 1